universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Deadline-Aware Resource Allocation and Scheduling for Serverless Workloads in Heterogeneous Clusters

verfasst von | submitted by

Matthias Fritz BSc

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 066 921
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von | Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Siegfried Benkner
Mitbetreut von | Co-Supervisor:	Dipl.-Ing. Dr.techn. Enes Bajrovic BSc

Acknowledgements

I would like to express my sincere gratitude to Siegfried Benkner and Enes Bajrovic for their invaluable guidance throughout my research journey. I greatly appreciate the opportunity to work with them over the years as part of the research group, which has helped me explore my research interests and deepen my knowledge in this field.

Finally, I want to thank my family and friends for their continuous support and encouragement, which has been a great source of motivation throughout this period of my life.

Abstract

Serverless computing has become widely adopted as a cloud execution model due to its ease of use and fine-grained pay-as-you-go pricing model. By abstracting away infrastructure management, it simplifies access to cloud resources, allowing developers to better focus on writing application code. However, most serverless platforms operate on a best-effort basis and offer limited flexibility when it comes to tuning resource allocations for performance. In addition to the lack of insight into the underlying hardware, it becomes increasingly difficult to reliably achieve Service Level Objectives (SLOs). However, the inherent heterogeneity of cloud environments also provides significant opportunities to deploy workloads with diverse resource requirements at different performance and price points. To address this hardware and performance variability, this thesis introduces DHRT, a deadline- and heterogeneity-aware scheduling and resource allocation framework for performance-critical serverless workloads. To ensure compatibility across different environments and cloud providers, DHRT utilises Knative, an open-source serverless platform built on Kubernetes. Applying online optimisation techniques based on heuristics, DHRT iteratively refines its resource estimates according to workload requirements and the available node heterogeneity by leveraging real-time metrics and historical execution data from live workloads. Using synthetic workloads, evaluations are conducted within the Google Kubernetes Engine to evaluate DHRT against baseline resource allocation and scheduling policies commonly employed by cloud providers for Function-as-a-Service (FaaS) offerings. The results show that DHRT can accurately learn resource demands of workloads from just a few live execution samples, eliminating the need for extensive manual resource tuning. Compared to baseline methods, it effectively exploits node heterogeneity for more targeted resource allocation and scheduling decisions, therefore achieving higher resource efficiency. Additionally, by proactively increasing CPU allocations as workloads approach their deadlines, DHRT is able to significantly reduce the risk of deadline violations.

Kurzfassung

Serverless Computing hat als benutzerfreundliche Alternative zu traditionellen Cloud-Lösungen immer mehr an Bedeutung gewonnen. Durch die Abstraktion und Automatisierung der Infrastrukturverwaltung ermöglicht es Entwicklern, Anwendungen einfach und kosteneffizient in der Cloud auszuführen. Allerdings bieten aktuelle Serverless-Dienste kaum Möglichkeiten, Ressourcen gezielt an die Leistungsanforderungen von Anwendungen anzupassen. Zusätzlich zur fehlenden Transparenz bezüglich der eingesetzten Hardware wird es zunehmend schwieriger, Service-Level-Objectives (SLOs) zuverlässig einzuhalten. Gleichzeitig könnte die Hardwarevielfalt in modernen Cloud-Umgebungen bedeutende Chancen für Entwickler bieten, um Anwendungen mit unterschiedlichen Ressourcenanforderungen effizienter zu betreiben. Um diese Hardware- und Leistungsvariabilität zu berücksichtigen, stellt diese Masterarbeit DHRT vor, ein Framework zur automatischen Ressourcenverwaltung, das eine zuverlässigere Einhaltung von SLOs für leistungskritische Serverless-Workloads ermöglicht. Um eine breite Kompatibilität zwischen verschiedenen Umgebungen und Cloud-Anbietern zu ermöglichen, nutzt es Knative, eine Open Source Serverless-Plattform basierend auf Kubernetes. DHRT setzt Online-Optimierungen basierend auf Heuristiken ein, um die Ressourcenzuweisung von Anwendungen dynamisch anhand von Echtzeitdaten anzupassen. Dabei integriert das System kontinuierlich Daten vergangener Ausführungen, um die Verwendung von Ressourcen iterativ zu verbessern. Dadurch kann es spezifische Workload-SLOs sowie die Heterogenität der verfügbaren Hardware gezielt berücksichtigen. DHRT wird in der Google Kubernetes Engine mit synthetischen Workloads gegen Referenzmethoden evaluiert. Die Ergebnisse zeigen, dass DHRT den Ressourcenbedarf von Workloads bereits nach wenigen Ausführungen erlernen kann und dadurch den Aufwand manueller Ressourcenanpassungen deutlich reduziert. Im Vergleich zu den Referenzmethoden kann DHRT die verfügbaren Ressourcen effizienter nutzen, da die Heterogenität der Hardware gezielt in den Ressourcenoptimierungsprozess integriert wird. Außerdem kann das System das Risiko von SLO-Verletzungen erheblich senken, indem CPU-Zuweisungen dynamisch angepasst werden, wenn sich Workloads ihrer Deadline nähern.

Contents

Acknowledgements i

Abstract iii

Kurzfassung v

1. Introduction 1
- 1.1. Motivation . . . 1
- 1.2. Problem Statement . . . 2
- 1.3. Goal and Contribution of the Thesis . . . 3
- 1.4. Structure of the Thesis . . . 3

2. Background 5
- 2.1. Technology Stack . . . 5
- 2.2. Terminology . . . 6
- 2.3. Containerization . . . 7
- 2.4. Kubernetes . . . 7
  - 2.4.1. Cluster Architecture Overview . . . 7
  - 2.4.2. Pod Resource Configuration . . . 8
  - 2.4.3. Pod Scaling . . . 9
  - 2.4.4. Kubernetes Scheduler . . . 10
  - 2.4.5. Kubernetes Scheduling Framework . . . 11
  - 2.4.6. Feature Gates . . . 13
- 2.5. Serverless Computing . . . 14
- 2.6. Knative: Serverless on Kubernetes . . . 14
  - 2.6.1. Knative Serving: Deploying Serverless Workloads . . . 15
  - 2.6.2. Autoscaling . . . 17
  - 2.6.3. Queue Proxy . . . 18
- 2.7. Resource Heterogeneity . . . 18
  - 2.7.1. Google Cloud Resource Taxonomy . . . 18

3. Related Work 21
- 3.1. Performance and Cost Optimisations in Serverless Computing . . . 21
  - 3.1.1. SLO-Aware Workload Scheduling . . . 22
  - 3.1.2. SLO-Aware Resource Configurations . . . 24
- 3.2. Performance and Cost Optimisations in Heterogeneous Computing Environments . . . 25
  - 3.2.1. SLO-Aware Scheduling of Workloads . . . 27

4. Methodology **29**
4.1. System Model . . . 29
4.1.1. Input and Optimisation Objective . . . 29
4.1.2. Resource Optimisation Process . . . 30
4.2. Scope and Limitations . . . 31
4.2.1. Optimisation Scope . . . 32
4.2.2. Workload Characteristics and Assumptions . . . 32
4.2.3. Resource Heterogeneity . . . 32
4.3. Experiment Set-Up . . . 33
4.3.1. Cluster Configuration . . . 33
4.3.2. Experimental Workloads . . . 34
4.3.3. Addressing Cluster Performance Variability . . . 34
4.3.4. Load Testing Framework . . . 35
4.4. Data Collection and Evaluation Strategy . . . 35

5. Implementation of DHRT **37**
5.1. System Overview . . . 37
5.1.1. System Workflow . . . 38
5.2. Heterogeneity-Aware Scheduling . . . 40
5.2.1. PreFilter Phase . . . 40
5.2.2. Filter Phase . . . 41
5.2.3. Score Phase . . . 41
5.2.4. PreBind Phase . . . 45
5.2.5. Deployment and Usage of HDS . . . 45
5.3. In-place Resource Tuning of Pods . . . 46
5.3.1. Tracking Resource Updates with Kubernetes Informer . . . 47
5.3.2. Timing and Frequency of In-Place Pod Resizing . . . 47
5.3.3. Pod Resource Evaluation . . . 47
5.3.4. Vertical Scaling Strategy . . . 48
5.3.5. Reactionary Scaling for Long-Pending Pods . . . 50
5.3.6. Maintaining Up-To-Date Resource Estimates . . . 50
5.3.7. Performing In-place Pod Resource Updates . . . 51
5.4. Data Processing and Heuristic-based Resource Estimation . . . 51
5.4.1. Webhook-Driven Workflow . . . 51
5.4.2. Data Retrieval and Aggregation . . . 52
5.4.3. Optimisation of Resource Configurations . . . 53
5.4.4. Integrating Node Pools for Resource Estimation . . . 55
5.4.5. Updating a Pod's Resource Profile . . . 56
5.4.6. Interplay with IRT . . . 57
5.5. Monitoring and Observability . . . 57
5.5.1. Metric Collection using Prometheus . . . 58
5.5.2. Log Aggregation - Loki . . . 58
5.5.3. Observability - Grafana . . . 59

5.6. Deterministic Pod Termination in Knative Serving . . . 60
5.7. System Deployment and Usage . . . 60
5.7.1. Workload Deployment . . . 61

6. Evaluation **63**
6.1. Overview . . . 63
6.2. Resource Allocation and Deadline Adherence within Homogeneous Clusters 64
6.2.1. Fixed CPU-Allocation Policy . . . 65
6.2.2. Proportional CPU-Allocation Policy . . . 66
6.2.3. DHRT: Deadline-Aware Resource Allocation Strategy . . . 68
6.2.4. Summary of Findings . . . 71
6.3. Heterogeneity-Aware Scheduling and Resource Adaptation . . . 72
6.3.1. Results . . . 73
6.3.2. Discussion . . . 74
6.3.3. Summary of Findings . . . 76

7. Conclusion **79**
7.1. Contributions and Findings . . . 79
7.2. Challenges and Learnings . . . 80
7.3. Future Work . . . 80

List of Tables **83**

List of Figures **85**

Listings **87**

Bibliography **89**

A. Appendix **97**
A.1. Software Project . . . 97
A.1.1. Kubernetes Scheduler Configuration . . . 97
A.1.2. Queue-Proxy Request Logging . . . 98
A.1.3. InPlacePodVerticalScaling API . . . 99
A.1.4. Execution Profile Example . . . 100
A.1.5. HRE CPU Estimate Example . . . 101
A.1.6. Workload Requests and Responses . . . 103

1. Introduction

1.1. Motivation

Cloud computing has fundamentally changed infrastructure provisioning, making hardware more accessible than ever before. With on-demand resource scaling and a pay-as-you-go model, it offers a powerful alternative to traditional upfront hardware investments [1]. However, despite these advancements, effective management of Virtual Machines (VMs) and application deployment remains a challenging task, prompting developers to shift their attention increasingly towards operational responsibilities. While Platform-as-a-Service solutions built on Kubernetes were able to streamline many issues associated with infrastructure management, developers wanting to quickly deploy and run their workloads still found themselves looking for an easier path to the cloud [2].

These challenges motivated the emergence of serverless computing, which shifts operational responsibilities from the developer back to the cloud service provider. The most widespread form of serverless computing, known as Function-as-a-Service (FaaS), only requires developers to provide function code in a high-level programming language and specify events that trigger them. The cloud provider then handles the resource management and application lifecycle, including seamless scaling to meet ongoing demand. Moreover, customers are only charged based on the actual resources used, rather than paying a fixed rate for the reservation of resources [2].

However, this ease of use inherently comes at the cost of reduced control over workload execution. Typically, serverless providers limit developers to choose from a range of memory increments to configure their function. Based on this selection, the vCPU allocation is then made either proportionally or with a static configuration of a single vCPU. Translating these resource configurations to real-world performance can be quite challenging, especially as the number of deployed functions grows. This becomes even more difficult due to the lack of predictable performance, as developers do not have any insight into the underlying hardware used, which can change from request to request [3].

Given this best-effort model, deploying performance-critical workloads is impractical, as Service Level Objectives (SLOs), such as execution time targets, cannot be reliably met. With workloads having increasingly diverse resource requirements, aligning this heterogeneity with the underlying hardware presents significant opportunities. For example, using a mixture of cost-efficient and compute- or memory-optimised nodes could offer a balanced approach to maintain diverse performance requirements at a cost-efficient level.

To address these challenges, a deadline- and heterogeneity-aware function placement algorithm and resource allocation scheme are needed, which consider both individual workload demands as well as available node capabilities. By moving towards an automated and SLO-driven serverless deployment model, developers could benefit from a more intuitive and streamlined experience to deploy performance-critical workloads.

1.2. Problem Statement

Existing FaaS offerings provide very limited support for workload SLOs, which are often restricted to general metrics like service availability or resource capacity guarantees. However, customers usually prefer specific and directly measurable guarantees that accurately reflect the real-world performance of their applications [4]. These limited configuration options combined with the lack of information regarding the utilised hardware make it nearly impossible to reliably achieve performance targets.

Transitioning from managed serverless solutions to open-source platforms deployed directly on Kubernetes, like Knative, can provide greater flexibility when it comes to node selection and resource allocation. Nevertheless, current schedulers are inherently homogeneous and fail to recognise the diversity found in the underlying hardware, often leading to inefficient usage patterns. Additionally, there is typically minimal information available about resource requirements, which can result in performance degradation due to interference between co-located workloads [5, 6]. While options exist to fine-tune scheduling decisions and resource allocations, they impose significant developer overhead, requiring in-depth knowledge of both workload characteristics and available hardware. This fundamentally contradicts the ease of use promise of serverless computing. Moreover, these options tend to be too static and fail to capture individual workload SLOs.

To enable an SLO-driven serverless deployment model, many challenges need to be addressed. Since workload characteristics and invocation patterns are typically unknown beforehand, an online optimisation approach is necessary. This involves dynamically matching node capabilities with workload requirements while considering the current utilisation of nodes and user-specified SLOs. Additionally, to maintain performance targets without compromising on resource efficiency, resource allocations must be dynamically fine-tuned and iteratively improved for subsequent executions. This requires access to real-time monitoring information, with collected data continuously feeding into future resource allocation and scheduling decisions.

Heterogeneous clusters introduce additional complexity due to differences in CPU platforms, machine generations and varying configurations of virtual resources, such as compute- or memory-optimised nodes. These differences, combined with the inherent performance variability in virtualised cloud environments, affect resource efficiency and introduce uncertainty in workload execution times. Therefore, incorporating heterogeneity

into the resource optimisation process is essential to achieve both reliable performance and efficient resource utilisation.

1.3. Goal and Contribution of the Thesis

The goal of this thesis is to explore how the scheduling and resource allocation of serverless workloads can be optimised within heterogeneous compute clusters to reliably achieve SLOs. The primary focus lies on performance-driven and compute-bound workloads, which adhere to strict runtime deadlines.

The contribution of this thesis is the development of Deadline- and Heterogeneity-aware Resource Tuner (DHRT), a system that automates and iteratively fine-tunes resource management for workloads. In principle, it utilises online optimisation techniques based on heuristics to dynamically adjust resources allocations and scheduling decisions based on workload demands and cluster conditions. The main objective is to ensure user-specified runtime deadlines are met while maintaining high resource efficiency. Additionally, DHRT accounts for hardware heterogeneity, ensuring that available resources are effectively utilised according to workload characteristics.

Based on these objectives, the following research questions guide this thesis:

RQ1. How effectively can in-place resource tuning based on real-time metrics mitigate performance variations and reduce deadline violations?

RQ2. To what extent can historical data and heuristics improve resource estimates to efficiently meet runtime deadlines?

RQ3. How can scheduling decisions and resource allocations be improved by incorporating node heterogeneity?

RQ4. How does the developed approach compare to conventional schedulers and resource allocation policies in terms of deadline adherence and resource efficiency?

1.4. Structure of the Thesis

This thesis is structured as follows: **Chapter 2** provides background information on key concepts and technologies relevant to this thesis. **Chapter 3** showcases related work concerning different optimisation techniques in the fields of serverless and heterogeneous computing. The methodology, including system design, scope and limitations, experimental setup and evaluation strategy, is detailed in **Chapter 4**. **Chapter 5** demonstrates and discusses the implementation of DHRT and its individual components. Subsequently, the evaluation of the developed system to answer the research requestions is presented in **Chapter 6**. Lastly, **Chapter 7** concludes the thesis, summarising key findings and providing an outlook for future work.

2. Background

2.1. Technology Stack

The developed system relies on a wide variety of different technologies, including programming languages, deployment platforms as well as monitoring and datastore solutions. A comprehensive overview is presented in Table 2.1. Subsequently, core concepts central to this thesis are described in more detail.

Category	**Technology**	**Description**
Programming Language	Go	Open-source and lightweight programming language supported by Google. Widely used in the development of Kubernetes tools and libraries. [`https://go.dev`]
Programming Language	Python	High-level, general-purpose programming language. [`https://www.python.org`]
Data Storage	MongoDB	Open-source Document-oriented NoSQL database system for highly scalable applications. [`https://www.mongodb.com`]
Monitoring	Prometheus	Open-source monitoring system and time series database. [`https://prometheus.io`]
Monitoring	Grafana	Open-source analytics and monitoring platform. [`https://grafana.com`]
Log Aggregation	Loki	Open-source log aggregation system for collecting, storing, and querying logs. [`https://grafana.com/oss/loki`]
Containerization	Docker	Open-source containerization platform to package software in lightweight containers. [`https://www.docker.com`]
Container Orchestration	Kubernetes	Open-source container orchestration platform originally developed by Google. [`https://kubernetes.io`]
Serverless Platform	Knative	Open-source and Kubernetes-based solution to build serverless and event-driven applications. [`https://knative.dev`]
Load Testing	Locust	Open-source tool for scalable load testing using flexible Python scripts. [`https://locust.io`]

Table 2.1.: List of core technologies utilised for the developed system.

2.2. Terminology

In the context of this thesis, several terms are frequently used which may have a different meaning depending on the context used. To ensure clarity and consistency, these terms are defined below:

Workload In general Kubernetes terminology, a workload refers to "an application running on Kubernetes" [7]. However, in the context of this thesis, which focuses on serverless execution, a workload specifically represents a single invocation of a program. Each invocation runs in an isolated Pod, executes a task and finally returns a result.

(Runtime) Deadline The maximum allowable time for a workload execution to complete and return a response. If this threshold is exceeded, the execution is considered a deadline violation.

Resource Efficiency Refers to minimising resource waste by allocating only the necessary resources for a workload to terminate before its deadline.

Over-Provisioning Allocating more resources than what is actually required by the workload, resulting in inefficient resource utilisation and increased costs.

Under-Provisioning The allocation of insufficient resources for a workload, which can result in performance degradation and deadline violations.

Resource Fragmentation Available resources (CPU, memory) are scattered in a way that prevents workloads from being scheduled because no single node has enough contiguous resources to accommodate them.

Resource Imbalance A situation where one resource (e.g. CPU or memory) on a node is underutilised while the other is highly utilised or exhausted, therefore limiting scheduling flexibility.

Resource Request The minimum resources explicitly reserved to a container during its execution.

Resource Limit The maximum allowable resources that can be used by a container before it is throttled.

Resource Allocation The amount of CPU and memory assigned to a container, defined via Requests and Limits.

In-Place Pod Resizing / Vertical Scaling Adjust a Pod's resource allocation (CPU, memory) dynamically while it is running, without requiring a restart.

Resource Configuration The initial resource allocation defined during deployment, specifying the CPU and memory Requests for a container.

2.3. Containerization

Containerization is a fundamental technology for deploying and managing applications in modern cloud and distributed computing environments. The central concept of containerization is to package up the code of an application along with its dependencies into a lightweight and portable container image. Not only can this image be deployed quickly on any execution environment, it also behaves uniformly. By sharing the host's operating system, containers eliminate the need to run their own, let alone emulate virtual hardware when compared to traditional VMs. This operating system-level virtualisation allows containers to execute more efficiently, with container images remaining much smaller in size [8].

The most widely used containerization technology is Docker [9], which is also used throughout this thesis to package system components and workloads before deployment on Kubernetes.

2.4. Kubernetes

Kubernetes [10] is an open-source platform which has become the de-facto standard for container orchestration, with widespread support across cloud vendors. It has emerged as the predominant platform for modern deployments, streamlining many challenges associated with container management. This includes key stages throughout the container's lifecycle, ranging from its deployment, to scaling and load-balancing. The following sections provide a brief overview of Kubernetes' architecture and introduce central concepts relevant for subsequent chapters.

2.4.1. Cluster Architecture Overview

In short, Kubernetes clusters [11] consist of the control plane (also referred to as `master`) and a set of nodes, representing the worker machines that host the containerised workloads. An illustration of the cluster architecture is shown in Figure 2.1. The control plane oversees the entire cluster and handles tasks such as scheduling Pods, monitoring cluster health and managing the desired state of the system. This is done through key components such as the `API Server`, `Controller Manager`, `etcd` and finally the `kube-scheduler`, which is described in more detail in Section 2.4.4.

To execute containerised workloads, the control-plane places them onto worker nodes. In Kubernetes' abstraction model, the central unit of work is referred to as a `Pod`, which hosts one or more closely related containers. These containers share the same environment, including network and storage resources. The node agent, `kubelet`, then facilitates the coordination between the worker node and the control plane. Moreover, the `kube-proxy` is typically deployed on every node, which manages network rules to enable communication between Pods and services [11].

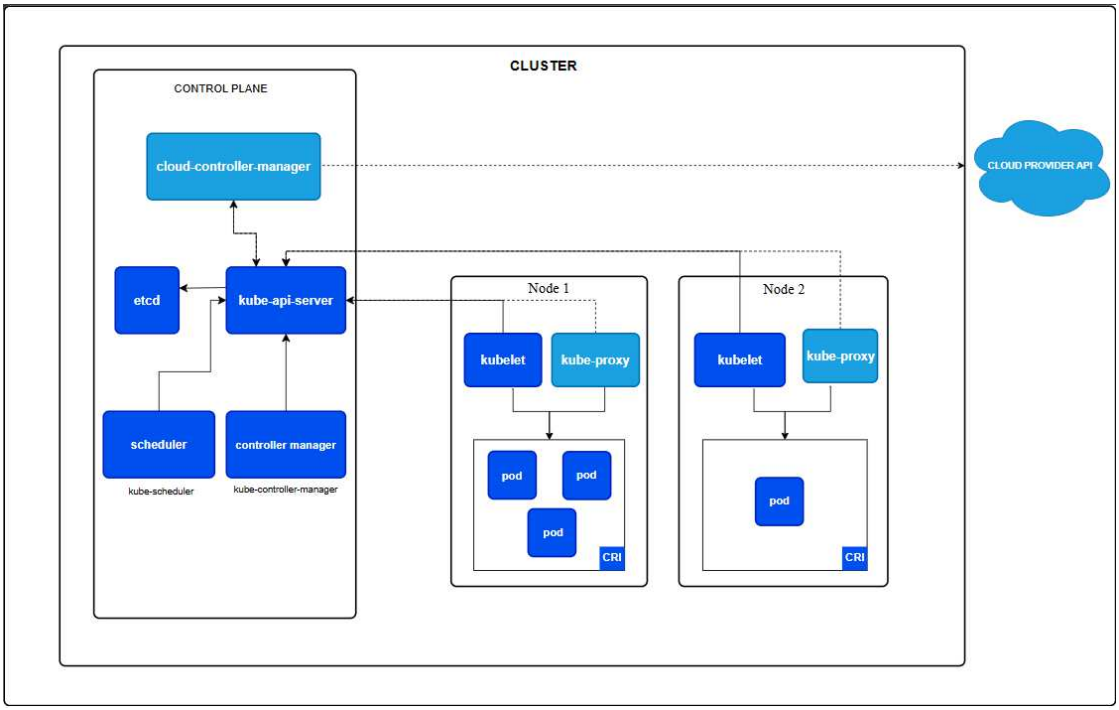

Figure 2.1.: **Kubernetes Cluster Architecture.** The example cluster consists of the control plane and two worker nodes, each hosting a set of Pods. The cloud-control-manager as well as the kube-proxy can be optionally deployed [10].

2.4.2. Pod Resource Configuration

Within a Pod's specification, developers can configure the CPU and memory that each container requires to operate. These are split into two key parameters: `Resource Requests` and `Resource Limits` [12]. The `Resource Request` specifies the resources that are reserved by the kubelet and guaranteed throughout the Pod's execution. This value is particularly important for the kube-scheduler, as it is a fundamental metric for its scheduling decisions. An example of how these resource configurations can be set for individual containers is demonstrated in Listing 2.1.

To efficiently use resource that are currently idle, Pods may also use more resources than requested. This can be useful to accommodate short-term usage spikes without having to explicitly reserve these resources for the entire Pod's lifecycle. Nevertheless, significantly exceeding the requested resources can lead to performance degradation for other Pods due to resource contention. Therefore, it is important to specify `Resource Limits`, which define the maximum amount of resources that a Pod is allowed to consume. With Limits in place, the kubelet will throttle the Pod whenever it approaches the specified threshold in order to facilitate more stable and fair resource sharing [12].

Lastly, it should be noted that these resource configurations are rather inflexible, requiring a redeployment of the Pod if either the Requests or Limits are changed. Nevertheless, Kubernetes has addressed this limitation with a feature currently in alpha stage. Being an important part of the proposed system, this experimental feature is further described in Section 2.4.6.

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: my-app:latest
    resources:
      requests:
        memory: "512Mi"
        cpu: "500m"
      limits:
        memory: "1024Mi"
        cpu: "1000m"
  - name: metrics-exporter
    image: prom/prometheus:latest
    resources:
      requests:
        memory: "64Mi"
        cpu: "50m"
      limits:
        memory: "128Mi"
        cpu: "100m"
```

Listing 2.1: **Pod Manifest with Resource Requests and Limits.** The manifest configures CPU and memory Requests and Limits for both the user application and the sidecar container.

2.4.3. Pod Scaling

A key feature of container orchestration platforms such as Kubernetes is the automatic scaling of workloads to meet ongoing demand. In principle, there are two scaling strategies: horizontal and vertical scaling, which are briefly described below in the context of Kubernetes.

Horizontal Scaling

Horizontal scaling refers to scaling the number of replicas to divide incoming requests between multiple instances. Naturally, this requires that the application is designed to support such distributed execution. In Kubernetes, horizontal scaling is implemented via the Horizontal Pod Autoscaler (HPA) [13], which continuously updates the number of Pod replicas according to resource utilisation metrics. To achieve this, the developer

must deploy the HPA and configure a target metric, typically CPU or memory, as well as a utilisation threshold and optionally, the minimum and maximum number of replicas. The HPA then uses the Pod's Resource Requests as a baseline to automatically scale the workload based on real-time metric utilisation relative to the configured threshold.

However, determining good Resource Requests and scaling thresholds is not trivial and can require significant manual tuning. Additionally, the HPA does not account for node heterogeneity and treats all Pods equally. Lastly, the time it takes to provision a new replica may be insufficient for workloads with strict SLOs.

Vertical Scaling

In the context of Kubernetes, vertical scaling refers to the ability to resize the allocated memory and CPU of containers. While not built-in by default, Kubernetes offers a Vertical Pod Autoscaler (VPA) [13] which can be separately installed. Its main objective is to relieve the developer of configuring up-to-date Resource Requests and Limits for their containers based on their observed utilisation. However, these adjustments typically require Pod restarts and are primarily reactive, thus not aware of workload-specific SLOs.

For workloads sensitive to SLO constraints, vertical scaling offers improved responsiveness compared to horizontal scaling, especially when resource changes can be applied dynamically without disruptions. Therefore, a key part of this work relies on the experimental `InPlacePodVerticalScaling` feature to fine-tune resource allocation decisions. This feature is further described in Section 2.4.6 and its usage within DHRT in Section 5.3.

2.4.4. Kubernetes Scheduler

The `kube-scheduler` is a key component of Kubernetes' control-plane and responsible for placing Pods onto worker nodes. The scheduling process is divided into multiple phases, most importantly Filter and Score. During the Filter phase, nodes that are considered infeasible for scheduling, for example due to insufficient remaining resources, are removed as candidates for scheduling. Subsequently, the Scoring phase ranks each remaining node according to predefined criteria, where the node with the highest score is then selected to host the Pod [14].

The `kube-scheduler` is designed to be lightweight yet highly modular and extensible, with most of the scheduling logic being implemented through plugins. For instance, the `Node Affinity` plugin allows to prioritise certain nodes based on labels, enabling a more targeted placement of Pods. In contrast, if we want to prevent Pods from being scheduled on a node, a `Taint` can be applied to the node. This discourages or outright blocks scheduling, unless a Pod explicitly `Tolerates` the configured Taint [15].

Furthermore, several built-in plugins can be fine-tuned through the kube-scheduler configuration. For instance, the scoring strategy by default favours nodes with more

available resources for scheduling. However, this behaviour can be adjusted to prioritise the most allocated node instead to implement a bin-packing strategy [15].

Nevertheless, these default plugins are rather static when it comes to implementing more complex scheduling decisions. Thus, they may not be suitable for every use-case, especially for more dynamic scheduling requirements. Therefore, the next section will explain in detail how the scheduling logic can be extended by developing custom plugins.

2.4.5. Kubernetes Scheduling Framework

The Kubernetes Scheduling Framework [16] is an extensible architecture enabling developers to implement and incorporate custom logic directly into the scheduling process through its plugin Application Programming Interface (API). The framework defines multiple extension points where plugins can be registered and consequently triggered during different stages of the scheduling and binding phases of a Pod.

An overview of the framework's workflow and its exposed interfaces can be viewed in Figure 2.2. The individual phases and their respective roles according to the official documentation [16] are listed as follows:

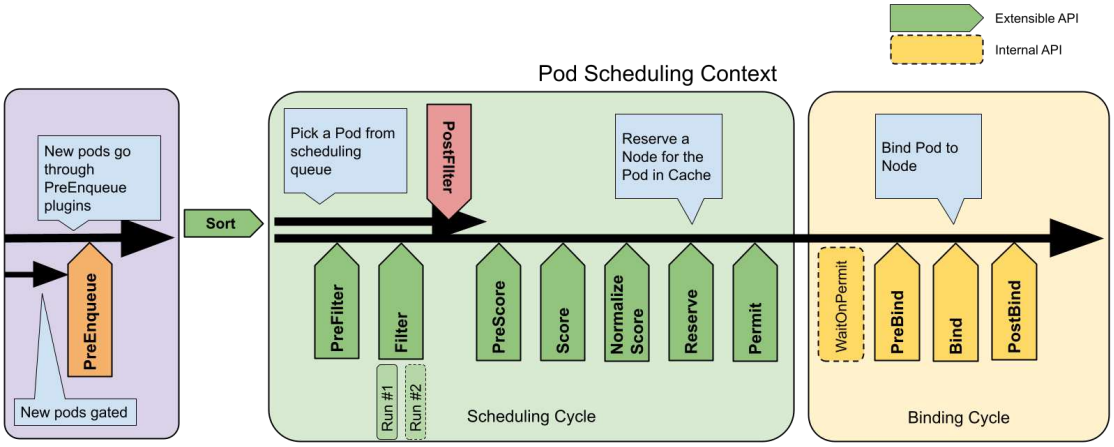

Figure 2.2.: **Kubernetes Scheduling Framework Overview.** The diagram showcases the individual extension points within the scheduling and binding cycle where custom plugins can be integrated [16].

1. **PreEnqueue**: Processes Pods before they enter the scheduling queue, allowing for early scheduling decisions or modifications. The Pod is permitted to enter the active queue only after all `PreEnqueue` plugins return a *success* status.

2. **Sort**: Determines the order in which Pods in the scheduling queue are processed based on predefined criteria or priorities, using a `Less(Pod1, Pod2)` function.

3. **PreFilter**: Evaluates Pods before the filtering phase to perform preliminary checks and store state information that can be accessed in later phases of the scheduling cycle. If an error is returned, the scheduling of the Pod is aborted.
4. **Filter**: Invoked for each available node to evaluate if it is a suitable candidate to host the Pod according to predefined criteria.
5. **PostFilter**: Triggered only when the `Filter` phase did not mark any node as suitable. Additional actions can be performed here to make scheduling still possible, such as preemption.
6. **PreScore**: Performs initial evaluations to generate a shareable state to avoid redundant computations and streamline the `Score` phase.
7. **Score**: Assigns scores to all nodes that have passed the `Filter` phase based on their suitability to host the Pod. The node with the highest computed score is then selected to host the Pod.
8. **Normalize Score**: If required, normalises the computed scores to ensure consistent comparisons before the final ranking of nodes is determined.
9. **Reserve**: Required resources are reserved on the selected node to ensure availability for the Pod.
10. **Permit**: Marks the end of the scheduling cycle and ultimately decides whether the Pod is allowed to be placed on the node, based on specified conditions or policies. In addition to approving or denying the Pod, it can return *wait* to issue a timeout and delay the binding phase.
11. **PreBind**: Performs checks or actions before the actual binding of a Pod to a node, such as provisioning a network volume.
12. **Bind**: Finalises the binding of the Pod to the selected node.
13. **PostBind**: An informational phase invoked after the Pod is successfully bound to the selected node. Primarily used for cleanup of temporary resources and data collection for analysis of the scheduling process.

Integration of Custom Plugins

To implement a custom plugin, it is recommended to refer to the official Scheduler-Plugins GitHub repository [17]. It documents and showcases the usage of the Scheduling Framework with official *out-of-tree* plugins which are not bundled with the kube-scheduler by default. These plugins can be integrated into the default kube-scheduler to modify or extend different scheduling phases. Alternatively, they can be combined into a standalone custom scheduler for more flexibility in the scheduling process.

Deploying custom plugins as a secondary scheduler is usually the more practical option, since modifying or replacing the kube-scheduler is not always feasible, especially in managed Kubernetes environments where direct access to the control plane is often restricted. Therefore, to improve portability across environments, it is generally recommended to run the custom scheduler alongside the default kube-scheduler. In this setup, Pods can specify the desired scheduler by setting the `schedulerName` property in their manifest file. This approach is also used in our scheduler implementation, which is described in Section 5.2.

2.4.6. Feature Gates

Feature Gates describe a set of configurable features in Kubernetes that can be turned on or off. These features are split into three different categories depending on their implementation maturity and stability [18]:

- **Alpha**: Disabled by default and their usage may introduce bugs. Support for the feature can be dropped and APIs may change significantly without notice. Not recommended for production clusters.
- **Beta**: Typically enabled by default and thoroughly tested. Support for the feature is expected to remain stable and its usage is considered safe. However, future changes may introduce incompatibilities, though migration instructions are provided.
- **Stable**: Also referred to as *General Availability* (GA). These features are enabled by default and no longer configurable through Feature Gates.

In-place Pod Resizing

An important feature to realise the proposed resource optimisation framework is `In-Place Pod Resizing` [19], which is controlled by the Feature Gate `InPlacePodVerticalScaling`. This experimental feature has first been introduced in Kubernetes version 1.27 and as of version 1.31, is still in alpha stage, requiring operators to explicitly enable the feature. Therefore, it is still under development and issues can occur during its usage.

This feature addresses the previously mentioned limitation that CPU and memory allocations (Requests and Limits) of a running Pod cannot be modified without causing a restart. In-place resizing enables dynamic adjustments to resource allocations, which allows to correct under- or over-provisioning in real-time. This provides a powerful mechanism to improve both performance and resource efficiency without disrupting the execution of active Pods. Moreover, it provides the option to temporarily boost allocated resources for expected short-term usage spikes, for example for initialisation-heavy applications [19]. An example of how the feature can be used via the `kubectl` CLI is demonstrated in Listing A.3.

However, changing the allocated resources during execution is not suitable for all types of applications. For example, databases can experience issues if the allocated memory is

suddenly decreased. To account for such cases, developers can specify a `restartPolicy` [19] for each resource. This policy determines whether the Pod should be restarted when the corresponding resource is resized.

2.5. Serverless Computing

Transitioning from a traditional, *serverful* deployment model to serverless is compared by Jonas et al. [2] to moving from assembly to a high-level programming language: Serverless has managed to overcome the complex and tedious management of cloud resources, effectively abstracting operational complexities from developers. It allows them to easily take advantage of cloud resources and deploy applications without the need for extensive infrastructure configuration and maintenance.

Where serverless fits within the cloud stack according to Jonas et al. [2] is illustrated in Figure 2.3. The serverless layer abstracts the virtual resources and services of the *Base Cloud Platform* to offer a set of high-level, managed solutions. The most well known serverless solution is FaaS, where developers must only provide their application code and specify an event that should trigger its execution. The required resources are then provisioned automatically when the function is invoked, with examples being Google Cloud Functions or AWS Lambda. However, serverless also includes Backend-as-a-Service solutions that provide managed infrastructure services such as databases, messaging systems or big data processing frameworks.

Nevertheless, while cloud providers offer highly sophisticated solutions, they often come with reduced flexibility and control due to higher levels of abstraction. Additionally, there is an increased risk of *vendor lock-in*, making it more challenging to migrate away from such proprietary ecosystems [2].

To address these limitations, platforms like Knative [20] and OpenFaaS [21] have emerged, which enable the deployment of serverless and event-driven applications directly on Kubernetes. By relying on open standards and integrating seamlessly with Kubernetes, they offer more portability and fewer restrictions, which reduces the dependency on proprietary cloud services. At the same time, such platforms still maintain the benefits of serverless, simplifying operational complexities and empowering application development.

2.6. Knative: Serverless on Kubernetes

Knative [20] is a platform-agnostic solution designed to enable the deployment of serverless workloads on Kubernetes. Originally developed by Google, the open-source project uses many open standards to facilitate the deployment of serverless applications in a highly portable way. It is designed to provide a clear separation of concerns between developers and operators by automating key operational tasks, including the deployment, traffic

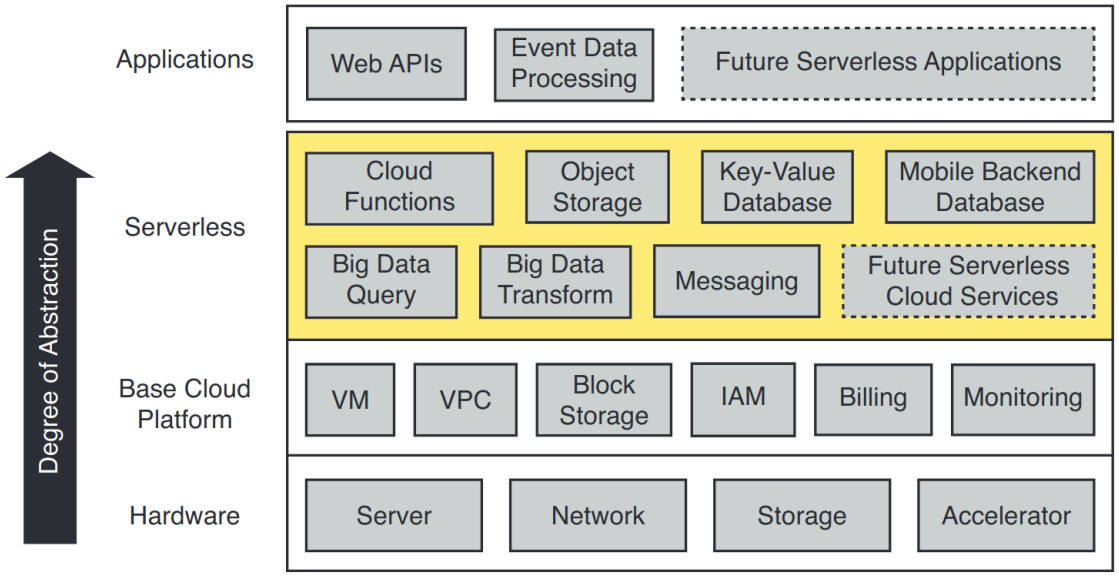

Figure 2.3.: **Cloud Computing Abstraction Layers.** Serverless provides an additional layer of abstraction over fundamental cloud resources. Within this serverless layer, cloud providers offer a wide range of managed solutions, including function deployment, data processing and various datastores [2].

management and autoscaling of containerised applications. Furthermore, Knative seamlessly integrates with Kubernetes, maintaining its inherent flexibility and extensibility to adapt to different infrastructure and application needs.

Knative consists of two modular components: Knative Serving [22], which enables the deployment and autoscaling of serverless workloads and Knative Eventing [23], which facilitates event-driven communication between services. Since this thesis focuses on the execution of serverless workloads rather than event-driven invocations, only Knative Serving is considered in more detail in subsequent sections.

2.6.1. Knative Serving: Deploying Serverless Workloads

Knative Serving [22] introduces its own custom Service object for deploying serverless workloads, which manages and coordinates all aspects of their lifecycle. To realise a serverless deployment model, Knative uses `Custom Resource Definitions` to extend the Kubernetes API and introduce custom resource objects. The primary components of Knative Serving can be viewed in Figure 2.4 and are subsequently described:

- **Configurations:** Configurations define the desired state of a Service, including the container image and environment variables. Each Configuration serves as a template for a Revision and thus, any modifications to it results in a new Revision.

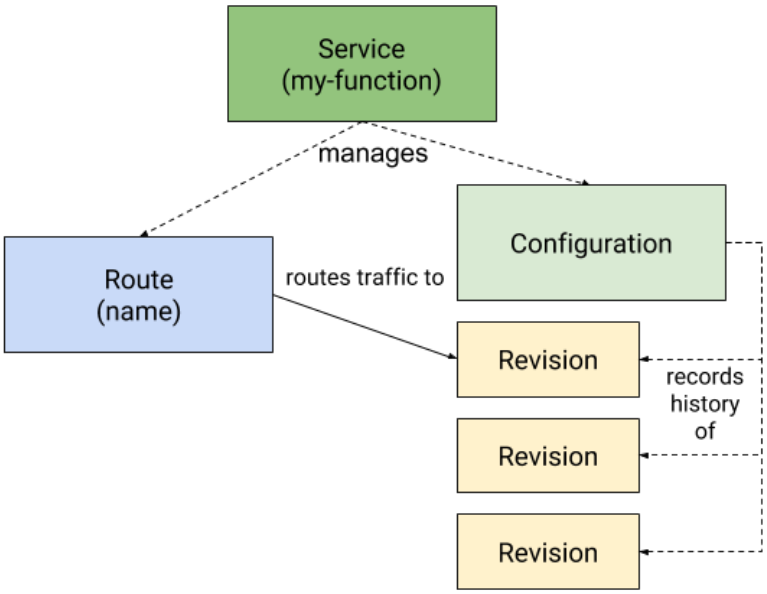

Figure 2.4.: **Knative Serving Components.** A Service object manages a configuration, which defines the desired state of the Service. Additionally, it configures a network route to allow traffic to reach Revisions [22].

- **Revisions:** A Revision represents a snapshot of a state defined by a Configuration and can be considered a version history of a Service. Traffic can be split between multiple revisions to enable sophisticated traffic management and continuous deployment strategies.
- **Routes:** A Route provides a network endpoint to route traffic to Revisions.

Example Serving Deployment

To demonstrate the practical use of Knative Serving, an example Service manifest is presented in Listing 2.2. Using just a short description of key metadata, developers can deploy their applications while Knative then handles operational complexities, including Pod creation, automatic network routing and dynamic autoscaling. The example highlights the simplicity and efficiency of the serverless deployment model, which automates time-consuming operational tasks and allows developers to focus on application logic.

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    metadata:
```

```
9       annotations:
10        autoscaling.knative.dev/min-scale: "0"
11        autoscaling.knative.dev/max-scale: "5"
12        autoscaling.knative.dev/metric: "concurrency"
13        autoscaling.knative.dev/target: "3"
14    spec:
15      containers:
16      - image: example-service:latest
17        ports:
18        - containerPort: 8080
```

Listing 2.2: **Example Knative Service Manifest.** The shown manifest file defines a Knative Service configured to run the latest container image of `example-service`. The application exposes traffic on port 8080 and is configured to scale between 0-5 Pod replicas, using a target concurrency of 3 requests as a scaling baseline.

Knative Functions

Knative also provides a simpler deployment model which is more in line with FaaS solutions and does not require any knowledge about Kubernetes or containers. Knative Functions [24] essentially adds another layer of abstraction on top of Knative Serving, further simplifying the deployment of functions. It provides function templates for popular high-level programming languages and automatically builds a container image from the function code. The created function can then be directly deployed to the cluster as a Knative Service and subsequently invoked.

2.6.2. Autoscaling

One of the key features of serverless platforms is their ability to automatically scale applications based on demand, even down to zero when no traffic is present. In Knative, horizontal scaling is achieved through the Knative Pod Autoscaler (KPA), which dynamically adjusts the replicas of a Pod based on traffic. Key parameters, such as `Concurrency` and `Scale Bounds`, can be configured to control the scaling behaviour. Unlike Kubernetes' traditional Horizontal Pod Autoscaler, which relies primarily on CPU and memory utilisation metrics, Knative adjusts the number of Pods based on Hypertext Transfer Protocol (HTTP) request metrics by default. This provides a more intuitive and application-centric scaling approach to developers [22, 25].

Based on the defined target metrics, the KPA continuously monitors the current request load and updates the number of replicas accordingly. This is achieved through a control loop that dynamically updates the underlying Kubernetes resources, specifically the Deployment and its ReplicaSet. When traffic exceeds the scaling threshold, the KPA triggers an update to increase the number of Pod replicas accordingly. Conversely, if traffic decreases, the KPA reduces the replicas to optimise resource usage. The goal is

to ensure that only the necessary resources are allocated to handle incoming requests efficiently [22].

2.6.3. Queue Proxy

A key component in Knative is the Queue-Proxy [20], which is deployed as a sidecar container in every Pod alongside the user application. It is responsible for forwarding incoming HTTP traffic to the user-container and enforcing concurrency limits by buffering incoming requests. In addition to serving as a reverse proxy, it collects and reports valuable request metrics, such as the number of concurrent requests or requests per second. These metrics are essential to facilitate autoscaling based on HTTP-traffic.

Queue-Proxy also plays a crucial role in the developed system to capture fine-grained request metrics and to control the behaviour of Pod deletions, which is further described in Section 5.4.1 and Section 5.6 respectively.

2.7. Resource Heterogeneity

The term *Heterogeneity* within the scope of compute clusters can be quite broad. On one hand, it can refer to the inclusion of hardware accelerators within clusters, such as GPUs or TPUs. On the other hand, it can describe different configurations of machines in terms of CPU, memory or storage. These variations can exist at the physical level, such as different CPU platforms and generations, which naturally occurs as servers are replaced over time. However, heterogeneity can also be found on a virtual level, where nodes can be allocated varying amounts of vCPUs or memory [5, 6].

In the context of this thesis, we focus on heterogeneous clusters as the latter, consisting of differently equipped standard compute nodes, both in terms of physical and virtual heterogeneity. For example, using clusters that consist of a mixture of balanced, cost-efficient nodes alongside compute- or memory-optimised ones. To provide some insight into the range of available node types and configurations in public clouds, a brief overview of Google Cloud's offerings is presented subsequently.

2.7.1. Google Cloud Resource Taxonomy

In general, Google Cloud categorises its available machines into different families, each optimised for specific purposes. The following list, based on the official documentation [26], provides an overview of available *Machine Families*:

- **General-purpose:** Offer a balance of compute, memory, and storage resources for a variety of workloads with the best price-performance ratio.
- **Storage-optimised:** Ideal for workloads with low CPU usage but high storage requirements.

- **Compute-optimised:** Focus on compute-intensive tasks, offering high-performance CPUs with a higher vCPU-to-memory ratio.
- **Memory-optimised:** Provide high memory-to-vCPU ratios for memory-intensive workloads.
- **Accelerator-optimised:** Integrate GPUs or TPUs for hardware-accelerated workloads such as machine learning and high-performance computing tasks.

Within these Machine Families, distinct *Machine Series* are offered, representing different hardware generations and performance tiers. For example, within the compute-optimised (C) Machine Family, there are Intel-based `C2` and AMD-based `C2D` instances, where the number '2' indicates the hardware generation [26].

Beyond this, *Machine Types* provide further customisation, offering various configurations of vCPUs and memory. For example, within the general-purpose N1 series, there are `standard`, `highmem` and `highcpu` variants, each with a different vCPU-to-memory allocation ratio [26]. This flexibility allows customers to tailor node selection to their workload demands, as well as balance performance and cost efficiency.

3. Related Work

In this chapter we showcase existing work that explores strategies to facilitate efficient resource management within the scope of both serverless and heterogeneous computing. Fundamentally, both of these fields face similar challenges associated with resource allocation and workload scheduling. Given the potential to improve resource utilisation and reduce costs, significant research effort is focused on fine-tuning resource configurations and aligning workload demands with available compute capabilities. Such sophisticated resource management strategies can also help to enable the execution of workloads sensitive to SLOs. Thus, by examining differences and parallels of existing methods and frameworks, we can gain valuable insights for the development of a resource-aware serverless deployment framework tailored to SLO-driven workloads.

A concise overview of all reviewed contributions can be viewed in Table 3.1. The first half focuses on work in the area of serverless computing, while the latter half contains work within the domain of heterogeneous computing. The table outlines the authors' respective optimisation scope, namely scheduling, resource allocation or node selection and whether their method considers specific SLOs. Lastly, the key strategy employed by the individual contributions is briefly described.

3.1. Performance and Cost Optimisations in Serverless Computing

Serverless computing is still at an early stage but is quickly gaining popularity as a cost-efficient and developer-centric deployment model. Subsequently, many areas of research have emerged to explore its opportunities and challenges. A significant part of previous work [39–41] has been dedicated to developing strategies to mitigate cold starts, which occur due to the ephemeral nature of functions and elastic scaling properties of serverless environments. Other optimisation techniques include runtime optimisations based on the underlying hardware architecture [42] or *Function Fusion* approaches [43, 44] which attempt to determine the optimal size of serverless functions for more cost-efficient operations.

Contrary to these methods, we focus on research exploring strategies to optimise (1) the scheduling of serverless functions to nodes; (2) the allocation of memory and CPU to individual workloads. This is a non-trivial task which has significant cost and performance implications for both the cloud provider and the customer.

Current serverless providers usually do not consider workload characteristics during scheduling. Motivated by potential improvements to resource utilisation and costs, Mahmoudi et al. [27] present a machine-learning based approach to guide the placement of functions to available VMs. By leveraging data obtained from profiling on an isolated machine, their model aims to predict the normalised performance of workloads. Afterwards, they perform a second profiling step on a contested node to measure the performance impact due to co-location of other workloads on the same node. Consequently, by considering the current utilisation of VMs and the determined workload profile, their neural network based model tries to select the node that minimises performance degradation for deployed containers.

To evaluate the effectiveness of their smart spread algorithm, they deploy a mixture of CPU, memory and I/O intensive workloads within a small-scale homogeneous cluster. Comparisons to conventional function placement algorithms revealed throughput and response time improvements between 10-36% and 9-32% respectively. The authors emphasise that without taking the characteristics of individual workloads into account, it is not possible to guarantee an efficient placement of diverse workloads.

The options to configure resources for functions are inherently limited in serverless offerings in order to simplify the deployment for users. Therefore, Bilal et al. [3] investigate opportunities for performance and cost optimisations by fine-tuning not only memory and CPU allocations, but also leveraging different types of VMs. To achieve this, they conduct extensive benchmarks using six workloads with diverse resource requirements to systematically identify the best configuration across a very large search space. Their findings highlight the importance of sophisticated node selection and resource tuning algorithms, as some configurations result in up to a 14.9x longer runtime and 5.6x higher execution costs compared to the identified optimal setup.

Subsequently, they carried out experiments to compare the identified optimal configuration with resource allocation strategies commonly used by serverless providers. By contrast, these strategies are limited to a single instance type and offer varying degrees of flexibility regarding the allocation of CPU and memory. Their findings highlight that utilising different instance types can significantly improve function execution times. On the other hand, decoupling and fine-tuning CPU and memory allocations can achieve more cost-efficient operations. Finally, to make the observed benefits more accessible in real-world settings, they explore the effectiveness of black-box optimisation algorithms to automatically discover good resource configurations.

3.1.1. SLO-Aware Workload Scheduling

Serverless offerings typically do not have any performance guarantees and are characterised by a best-effort model with significantly variable performance [45, 46]. However, there have been growing efforts to facilitate the deployment of performance-critical workloads by incorporating user-specified SLOs into the scheduling process.

Addressing this problem from the perspective of both the user and cloud provider,

Mampage et al. [28] present an approach targeting cost-effective utilisation of cloud resources without compromising on user-defined SLOs. Their goal is to find a balance between minimising the underutilisation of VMs to save costs while also avoiding performance degradation due to resource overloading. Thus, they propose a deadline-aware scheduler which attempts to efficiently allocate functions based on specified runtime-targets and the current utilisation of all active VMs. To achieve this, they first schedule functions heuristically based on the remaining deadlines of active functions and the available leftover resources. Subsequently, their *Dynamic Resource Alteration* scheme uses monitoring information to dynamically increase the allocated CPU resources of functions approaching their runtime deadline. This method not only counteracts suboptimal initial resource allocations, but also mitigates the impact of resource overloading by evicting tasks if necessary, which helps to prevent deadline violations.

Through experiments they achieved improvements of up to 25% in fulfilling application deadlines, while lowering resource consumption by up to three times compared to conventional scheduling policies. However, their work is limited to homogeneous computing environments, which potentially limits its ability to handle workloads with significantly different resource requirements without compromising on costs.

Suresh et al. [29] introduce *ENSURE*, a latency-aware scheduler that uses a greedy solution that prioritises packing requests on a single host before shifting to the next. The goal of this strategy is to prevent cold starts by scheduling requests on active hosts while allowing idle hosts to naturally scale down to conserve resources. Moreover, to reduce the impact of resource contention and prevent SLO violations, their framework dynamically adjusts the CPU-shares of applications as they approach their latency deadline. Additionally, to mitigate the impact of colds starts and maintain latency targets during high traffic periods, ENSURE proactively scales the number of hosts and containers.

Evaluations against three baseline schedulers demonstrated the robustness of their method, which was able to maintain latency targets by reducing cold-starts by up to 60% while requiring 18-52% fewer hosts. They attribute these results to being able to closely match the actual application load through proactive scaling, thus improving resource efficiency. However, a limitation of their work is that users are required to provide memory configurations for their workloads and do not have direct influence over the latency target. Instead, the target is determined through testing on an isolated node, making it rather inflexible and dependent on a specific node type. As a result, this could make it difficult to apply ENSURE in heterogeneous environments.

Nguyen et al. [30] present a framework to enable real-time serverless applications, such as video- or traffic monitoring. These applications are characterised by strict deadlines, where failure to meet them can lead to severe repercussions. However, serverless offerings typically operate on a best-effort model, which cannot reliably meet these requirements. To achieve such real-time guarantees, they highlight the importance of maintaining a guaranteed invocation rate, which refers to how quickly the application can acquire compute resources.

To allow users to specify real-time targets of individual functions, they extend the serverless interface with an additional attribute in the deployment specification. To achieve these guarantees, they develop a prototype that uses admission control to check whether a new function can be accepted, by considering its requirements and the present system state. This is complemented with a predictive container provisioning scheme to maintain real-time targets. Through initial experiments, they showcase that their prototype can successfully provide application quality guarantees when the real-time parameter is correctly tuned. However, challenges remain, such as dealing with sudden losses of requests resulting in temporary SLO violations.

3.1.2. SLO-Aware Resource Configurations

The memory configuration of functions is one of the few aspects that must still be specified by the user and is not handled by the FaaS provider. Typically, the amount of vCPU is allocated proportionally, making it difficult to find cost-efficient configurations which manage to achieve the desired performance objectives. To address this challenge, Akhtar et al. [31] introduce *COSE*, a framework designed to identify the best resource configuration for individual and chains of serverless functions. Through statistical learning, their goal is to find the memory configuration which minimises the incurred costs while maintaining customer objectives, such as runtime deadlines. They utilise execution traces to build a performance model that captures the relationship between cost/performance and function configurations. To achieve this, they use Bayesian Optimisation since it can adapt its search to find the most informative configuration samples, thus avoiding unnecessary samples. Moreover, by continuously incorporating newer execution traces into their model, it is designed to be resilient against variations in real-world performance. Experiments on commercial and simulated cloud environments showed that *COSE* was able to find a near-optimal solution within 15 samples in 95% of cases.

An alternative approach to determine SLO-conform memory configurations using performance modelling is presented by Safaryan et al. [32] who use a max-heap based optimisation algorithm. Instead of being limited to individual functions, they intend to extend this capability to serverless applications consisting of many functions. Their method creates artificial load across various memory configurations for each function and then builds an application call graph based on tracing data. Using this information, they build a performance model of the application's functions to estimate their performance with different memory configurations. Subsequently, by applying a max-heap based optimisation technique, they are able to efficiently determine the cheapest memory configuration that would satisfy application SLO requirements. Furthermore, their method can also incorporate different optimisation goals, such as the minimum overall cost or execution time.

Wen et al. [33] propose an adaptive resource configuration method specifically designed for serverless function workflows. They argue that especially for complex workflows consisting of a large number of functions, it becomes increasingly difficult to choose good resource configurations due to the large search space. To address this, they introduce *StepConf*, a

framework designed to automate the resource configuration of functions to meet end-to-end workflow SLOs. They highlight the importance of finding a well-performing balance between memory size and function parallelism. Specifically, they note that increasing memory size can be more advantageous for multi-core friendly functions, as this indirectly increases the number of vCPUs allocated. For other functions, performance can be improved by exploiting inter-function parallelism, which can be achieved by increasing the number of concurrent function instances.

In principle, StepConf uses an adaptive strategy to mitigate the impact of performance variations and reliably achieve SLOs. It performs dynamic adjustments to resource configurations at each function step in the workflow based on an online heuristic algorithm. However, it first needs to learn the relationship between different function configurations and their influence on the whole workflow's cost and performance. As a result, the applicability of their framework is limited to scenarios where developers provide function and workflow details beforehand to conduct offline performance estimations.

3.2. Performance and Cost Optimisations in Heterogeneous Computing Environments

An inherent limitation of most widespread scheduling approaches is that they fail to capture the heterogeneity of nodes, often resulting in degraded resource utilisation and performance. The following contributions showcase opportunities to improve performance and costs by incorporating the capabilities of nodes and workload characteristics into the scheduling process. Unlike previously reviewed work, these approaches are not confined to serverless environments. This changes the optimisation techniques, as there is typically more information available beforehand and workload executions tend to be less dynamic and ephemeral.

Mao et al. [34] attempt to address the limitation of the widely used spread algorithm, which simply assigns containers to the node with the fewest containers running. Thus, it fails to take the heterogeneity of nodes into account, which could have vastly different capabilities and resources available. To optimise the resources used on each node and mitigate the effects of resource contention, they develop *DRAPS*, a resource-aware placement scheme. First, it determines the dominant resource of each container using monitoring information. Afterwards, workloads with complementary resource demands are scheduled to the same node according to a heuristic algorithm. This allows them to maximise individual resources used on each node, such as CPU or memory. Furthermore, *DRAPS* employs a dynamic container migration strategy to mitigate resource bottlenecks on nodes as they arise.

Viswanathan et al. [35] present a cloud placement solution designed for private cloud environments. They emphasise that public clouds often hide many operational details from the user, which makes it difficult to deploy applications with strict resource requirements. As such, the goal of their method is to leverage the available node and workload information in private clouds to improve resource utilisation.

To begin, they simplify the scheduling process by focusing on the most dominant resource per workload. These resource profiles are then utilised in their placement logic to co-locate workloads with complementary usage patterns. However, they acknowledge that in practice, workloads often show significant performance variability. To address this, they prioritise allocating highly variable workloads to servers equipped with dynamic reconfiguration capabilities.

To improve the resource usage and achieve energy savings, Jivrajani et al. [36] propose a green scheduling approach using hierarchical clustering. Essentially, they attempt to simplify the matching of workloads to nodes by partitioning heterogeneous nodes into mostly homogeneous sub-clusters. Similarly, clustering is applied to workloads to identify and group jobs with similar resource demands. Based on these results, they apply a best-fit heuristic to identify the node cluster that best matches the resource profile of each job cluster.

Rocha et al. [5] present *HEATS*, a scheduler that is both heterogeneity- and energy-aware. One of its key features is that it allows users to define their preferred balance between performance and energy savings. Initially, an extensive probing phase is performed to evaluate the performance and energy characteristics of the available hardware resources and train a multiple linear regression model. This model is then used to estimate the performance and energy consumption of new workloads on each node type, based on their specified resource requirements. The resulting estimates are then used to deploy the task to the node that best matches the desired energy/performance ratio. Lastly, *HEATS* periodically recomputes the scheduling process based on real-time energy levels and resource usage. If a more suitable node is found, the workload is then migrated accordingly.

Operating on a large scale, a datacenter scheduler is proposed by Delimitrou et al. [6]. Their algorithm makes use of collaborative filtering to categorise incoming workloads based on information obtained from previously scheduled workloads and offline training of selected workloads on available hardware platforms. As a result, they are able to streamline the profiling process during the online phase. This allows them to minimise the overhead needed to classify incoming workloads and quickly estimate their heterogeneity and impact on co-located applications. The obtained classification is then utilised by a greedy scheduler to place workloads in a way to optimise both performance and resource usage.

Their experiments further highlight that by considering heterogeneity and interference, substantial improvements in resource efficiency and performance can be achieved. However, taking approximately a minute to schedule a workload, their approach might not translate well to more dynamic environments or latency-critical tasks.

3.2.1. SLO-Aware Scheduling of Workloads

Given the extensive choice of VM types and configurations available across cloud service providers, there is a significant challenge associated with resource selection. To address this, Thai et al. [37] propose a deadline-aware scheduler that aims to fulfil application performance targets at a cost-efficient level. They explore two options to strategically determine the best configuration of node types and quantities in a heterogeneous cluster: (1) find a global solution across all submitted tasks; (2) consider only a single job at a time to find local optima.

Through experiments they were able to demonstrate that while the former led to higher cost-savings, it required significantly more time than the second method. When compared to naive approaches using homogeneous clusters, they could observe cost savings from 4% to 31%. Additionally, by dynamically migrating workload in response to performance variations, they were able to reduce the occurrence of deadline violations in worst-case scenarios by 95%. However, since their work relies primarily on selecting the best nodes according to workload requirements, it requires extensive information beforehand and jobs to be submitted in large batches rather than dynamically.

Expanding beyond the cloud, Nastic et al. [38] introduce a scheduling framework to enable SLO-sensitive workloads in the Cloud-Edge-IoT Continuum. Due to the high dynamicity of edge environments, they require real-time information to maintain SLOs. To achieve this, they introduce two fundamental concepts: (1) a *Service Graph* which provides workload metadata and their interactions; (2) a *Cluster Topology Graph* which maintains the current node topology and Quality of Service parameters of network links between nodes. To then score available machines and dictate the scheduling, they primarily rely on locality properties.

For future work they plan to also consider other factors for scheduling, such as incorporating the inherent heterogeneity found in edge environments. This could allow the placement of workloads to machines that are particularly well-suited for it or to achieve energy efficiency or sustainability SLOs.

Work	Scope			Method Overview	
	S	RA	NS	SLO-Aware	Primary Strategy
Mahmoudi et al. [27]	✓	✓			Profiling and Neural Network based model to predict normalised performance of workloads
Bilal et al. [3]		✓	✓		Investigate the impact and automate the choice of resource allocations and node types
Mampage et al. [28]	✓	✓		✓	Dynamically adjust CPU allocation when nearing deadline-SLO; Dynamic task eviction
Suresh et al. [29]	✓	✓		✓	Dynamically adjust CPU-shares when nearing deadline-SLO; Proactive scaling to prevent cold-starts and resource contention
Nguyen et al. [30]	✓			✓	Admission control and predictive container provisioning to maintain invocation rate
Akhtar et al. [31]		✓		✓	Tracing to build performance model; Bayesian Optimisation to find resource allocation
Safaryan et al. [32]		✓		✓	Tracing to build performance model; Max-Heap based optimisation for resource allocation
Wen et al. [33]		✓			Find trade-off between memory size and function parallelism; Heuristic resource adjustments to mitigate performance variations
Mao et al. [34]	✓				Determine dominant resource and schedule workloads with complementary needs; Dynamic task migration
Viswanathan et al. [35]	✓				Align workload demands to node capabilities; schedule complementary workloads and consider performance variability
Jivrajani et al. [36]	✓				Hierarchical Clustering and best-fit heuristic to match workload demands to node capabilities
Rocha et al. [5]	✓				Profiling on different nodes to schedule based on specified energy/performance ratio
Delimitrou et al. [6]	✓				Collaborative filtering and profiling to estimate heterogeneity and interference of workloads
Thai et al. [37]	✓		✓	✓	Configure optimised heterogeneous cluster based on SLOs; Dynamic job-migration
Nastic et al. [38]	✓			✓	Model workload interactions and cluster topology in cloud-edge; Locality-aware scheduling

Table 3.1.: **Summary of reviewed contributions.** The table highlights the authors' respective optimisation scope (**S**: Scheduling; **RA**: Resource Allocation; **NS**: Node Selection), SLO-awareness and primary strategy employed.

4. Methodology

This chapter outlines the approach taken to address the Research Questions presented in this thesis. It is structured as follows: First, the proposed system model and the scope and limitations of this work are defined. Next, the experimental setup is described, including cluster configurations and the selection of workloads. Lastly, the data collection and evaluation strategy is outlined, highlighting key metrics to assess the effectiveness of the developed approach.

4.1. System Model

This section defines the design principles, and requirements that guide the implementation of DHRT. Figure 4.1 provides a high-level overview of the system model, illustrating its key operational steps and the primary data elements involved throughout a single workload execution.

4.1.1. Input and Optimisation Objective

The primary objective of this thesis is to develop a resource tuning framework that optimises the deployment of SLO-sensitive serverless workloads. Unlike traditional approaches, where resource allocation is static or manually configured, this system dynamically adjusts CPU and memory allocations of workloads based on user requirements and node-specific performance characteristics.

The system takes a service manifest file as input, which specifies:

- The workload's **container image** and metadata.
- A **runtime deadline**, representing the maximum time for execution to complete and return a response.

The primary SLO and central optimisation target of DHRT is to fulfil the user-specified runtime deadline while maintaining high resource efficiency. On one hand, this means minimising resource waste by allocating only the necessary resources for each workload to reliably meet its execution deadline. On the other hand, it means efficiently packing workloads onto nodes to minimise unusable resource gaps that prevent additional workloads from being scheduled.

To achieve these optimisation goals, two key aspects can be tuned. First, DHRT can adjust its scheduling decisions, prioritising different machines depending on workload

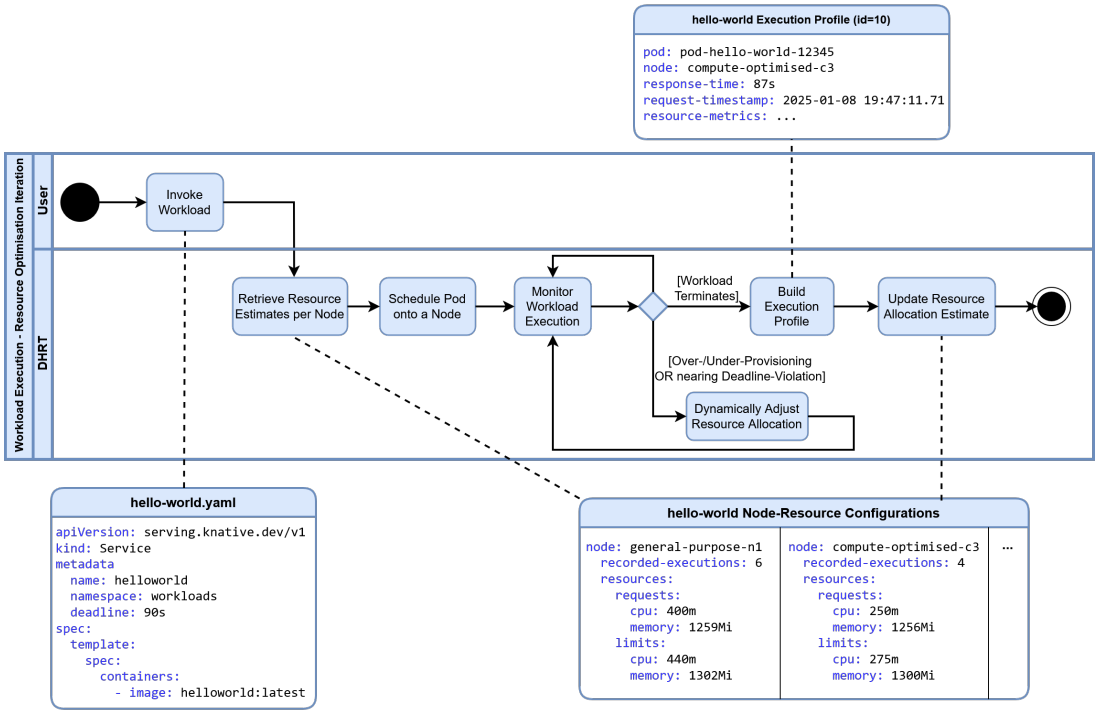

Figure 4.1.: **System Model Overview.** The figure illustrates the primary steps taken by DHRT for each workload invocation, including initial scheduling, real-time monitoring and dynamic in-place vertical scaling, with annotations showing the corresponding system data. As part of this online optimisation process, each iteration collects more workload data to continuously improve scheduling and resource tuning for future invocations. The only prerequisite is that the user deploys their workload as a Knative Service, as shown in the exemplary 'hello-world.yaml' manifest.

demands and current utilisation. Secondly, it can modify the amount of virtual resources allocated to a workload during its execution.

The impact of these decisions, especially in heterogenous environments and the employed strategies in the resource optimisation process are described in more detail in the following section.

4.1.2. Resource Optimisation Process

By designing DHRT to be aware of the available node heterogeneity, it allows developers to leverage a wide variety of nodes at different performance and price points. Additionally, it allows to fulfil SLOs for workloads with very diverse resource requirements. Therefore, integrating this heterogeneity into our resource tuning system presents significant opportunities to achieve improved performance and resource efficiency.

In terms of physical heterogeneity, a node with a more powerful CPU platform may execute a workload in the same time while requiring a lower vCPU allocation compared to a general-purpose node. At the same time, performance-optimised nodes can be prioritised for scheduling if lower-performance alternatives fail to meet the required execution time, even with increased vCPU allocations. This could be particularly relevant for sequential workloads that cannot easily benefit from additional vCPUs.

Virtual heterogeneity also plays an important role in workload placement, especially for workloads with imbalanced resource demands. These workloads may benefit from being placed onto complementary nodes, for example, scheduling a CPU-intensive workload with minimal memory requirements to a high-compute node which has a high vCPU-to-memory ratio. Taking this virtual heterogeneity into account can help to ensure resource availability and prevent resource bottlenecks.

To account for these resource variations, the system must dynamically discover node-specific resource allocations `[Memory, vCPU]` that reliably maintain execution deadlines. These resource configurations are derived from both workload characteristics and node capabilities. This not only relieves the developer of having to manually tune resources according to their SLOs, but also enables more targeted and efficient utilisation of available cluster resources.

To achieve these optimisation goals, three primary strategies are followed:

- **Dynamic Resource Allocation:** Adjust vCPU and memory allocations according to real-time metrics while the workload is executing and approaching its deadline.
- **Heuristic-Based Resource Estimation:** Utilise historical execution data and heuristics to predict workload resource requirements.
- **Adaptive Scheduling Decisions:** Select appropriate nodes based on real-time utilisation conditions, the resource demands of the workload and individual node capabilities within the heterogeneous cluster.

4.2. Scope and Limitations

As discussed in the reviewed contributions in Section 3, the fields of serverless and heterogeneous computing are very broad and encompass a wide range of optimisation techniques. These techniques vary from cloud-vendor focused approaches to user-centric optimisations that target different SLOs such as performance, cost-efficiency or reliability. Additionally, approaches differ in whether they are applied in an offline or online optimisation setting and in the types of workloads they address.

Given this broad scope, the following sections explicitly define the optimisation focus of this thesis, including its assumptions and limitations.

4.2.1. Optimisation Scope

This thesis focuses on online optimisation techniques for compute-bound, performance-critical serverless workloads. As previously described, the primary optimisation goal is to maintain user-specified execution deadlines. Naturally, since meeting performance targets could easily be achieved through over-provisioning, a secondary optimisation target is the efficient utilisation of available resources.

Since no prior information about workloads is available, optimisations must rely on real-time metrics and iterative learning from live executions. In contrast, offline optimisation methods, such as a-priori profiling of workloads or nodes, as well as methods focused strictly on cost reduction are outside the scope of this work.

4.2.2. Workload Characteristics and Assumptions

Workloads considered within this thesis are assumed to be input-independent and perform the same amount of computation across different invocations. Additionally, the execution time should be at least 60 seconds long to allow enough time for dynamic in-place adjustments and mitigate the impact of overheads like cold-starts or network delays.

The focus lies on CPU-bound workloads with varying memory requirements. This means that the execution time depends primarily on the underlying CPU platform and the allocation of vCPU to the workload. Naturally, parallel workloads offer more opportunities for resource tuning through vCPU scaling and are therefore of greater interest in this work. However, there are also resource optimisation opportunities for sequential workloads, particularly by leveraging heterogeneous nodes.

Each workload invocation is deployed as a separate container instance on a Pod using a concurrency setting of 1. This means that each request is handled in isolation without concurrent execution and resource-sharing within the same container. Nevertheless, identical workloads may be placed onto the same Pod given that the previous workload execution has finished and the Pod was not yet scaled down.

4.2.3. Resource Heterogeneity

A key aspect of this research is resource heterogeneity. However, this thesis does not explore heterogeneity in terms of specialised hardware accelerators, such as GPUs or TPUs. Instead, the focus is on heterogeneous clusters consisting of different compute nodes, both in terms of physical heterogeneity, such as different CPU platforms, but also virtual heterogeneity with varying vCPU and memory allocations.

Moreover, this work is limited to fixed-size clusters, where the cluster's capacity remains constant throughout its operation. Dynamically adding or removing nodes would introduce additional complexity and shift the focus towards cost optimisation and cluster

scaling strategies, which are beyond the scope of this thesis.

4.3. Experiment Set-Up

This section outlines the experimental setup used to evaluate DHRT. It describes the testing environment and the workloads used in the experiments. The objective is to provide sufficient context to interpret the obtained results and to ensure that the experiments can be reliably reproduced.

The test code along with deployment scripts and usage instructions can be found in the GitLab repository, as detailed in A.1.

4.3.1. Cluster Configuration

Evaluations are conducted within Google Kubernetes Engine (GKE) using both homogeneous and heterogeneous clusters. This allows to (1) isolate and analyse the impact of resource allocation without variability in nodes and (2) evaluate the influence of incorporating diverse node types and assess the impact of heterogeneity-aware optimisations.

The homogeneous cluster configuration, detailed in Table 4.1a, is comprised of general-purpose N1 nodes based on the Intel Skylake platform. The heterogeneous cluster builds on this by additionally incorporating a higher memory variant as well as a performance-optimised C3 node based on the Intel Sapphire Rapids platform. The exact configuration is described in Table 4.1b.

Machine Family	Category	vCPUs	Memory	Quantity
N1	balanced price & performance	4	7 GB	4

(a) Cluster Configuration A (Homogeneous Nodes)

Machine Family	Category	vCPUs	Memory	Quantity
N1	balanced price & performance	4	7.5 GB	2
N1	balanced price & performance	4	15 GB	1
C3	performance-optimised	4	8 GB	1

(b) Cluster Configuration B (Heterogeneous Nodes)

Table 4.1.: **Overview of cluster configurations used during the evaluation.** Cluster A utilises only a single type of node, whereas Cluster B introduces heterogeneity by incorporating different N1 configurations along with a performance-optimised C3 node. Detailed specifications of the machine families can be found in Google's official documentation [26].

4.3.2. Experimental Workloads

Evaluating the developed deadline-based resource management solution requires workloads with diverse resource and SLO requirements. Based on the workload specifications described in Section 4.2.2, a python-based implementation of the Power-Iteration algorithm is used for the subsequent evaluation experiments.

Given a Matrix `M`, this algorithm approximates its greatest eigenvalue and the corresponding eigenvector by repeatedly multiplying the vector v_k with `M` and normalising the result. This iterative step is defined as follows:

$$v_{k+1} = \frac{\mathtt{M} \cdot v_k}{\|\mathtt{M} \cdot v_k\|}$$

This workload was chosen because it effectively models resource-intensive applications and can be easily scaled to adapt its resource demands by adjusting its input parameters:

- `N:` Size of the $N \times N$ matrix.
- `Iterations:` Number of iterations performed to approximate the eigenvalue.

While the algorithm is primarily compute-bound, it can also become memory intensive as matrix sizes grow. Moreover, the specified user-defined deadline significantly impacts its resource needs, as tighter deadlines require proportionally higher vCPU allocations to meet performance goals. Since the workload uses NumPy matrix operations, it inherently benefits from parallel execution, which provides more opportunities for vCPU scaling to meet performance targets. These properties make it a great choice as it allows to easily model different resource requirements by changing any of the input parameters.

To illustrate the impact of different input parameters and resource configurations on the execution time and memory usage, Table 4.2 presents exemplary workload executions profiled on GKE.

4.3.3. Addressing Cluster Performance Variability

As experiments are conducted on virtualised hardware on GKE, there are a lot of factors that can influence performance, even if clusters are provisioned identically. To ensure more reliable comparisons, all experiments of the same type were executed on the same cluster. Additionally, a baseline execution time of the workload using 1 vCPU was established to derive the deadlines used in the evaluations.

Furthermore, since GKE varies the underlying CPU platform even within the same Machine Family, it can lead to noticeable performance differences between seemingly identical nodes depending on the current availability. Since we assume consistent performance within a homogenous node pool, only clusters were utilised where performance between equal machine types was comparable and stayed within minor performance variations.

Workload Input		Resource Configuration		Result Metrics	
N	**Iterations**	**Node Type**	**vCPUs**	**Execution Time (s)**	**Memory Usage (MiB)**
10 000	1000	N1	0.5	270	870
10 000	1000	N1	1	135	870
10 000	1000	N1	1.5	90	870
15 000	500	N1	1	152	1870
20 000	500	N1	1	264	3270
10 000	1000	C3	1	101	870
10 000	2000	C3	2	102	870

Table 4.2.: **Experimental Workload Metrics.** The table presents runtime information for the Power-Iteration workload to showcase its runtime behaviour under different input parameters and resource configurations. The data was obtained by testing on GKE in region 'europe-west-4'.

4.3.4. Load Testing Framework

To generate requests that invoke the experimental workloads, Locust [47] is utilised as it provides a lot of flexibility to define precise request patterns. It can schedule requests in highly customisable intervals, which allows to easily simulate different load scenarios. These requests can be launched asynchronously to ensure that each test creates the same amount of work, regardless of variations in response times.

This consistent test behaviour is essential to accurately evaluate and compare the performance of different methods. Additionally, Locust records important request metrics, such as request latency, throughput and failure rates, which are essential to assess SLOs.

4.4. Data Collection and Evaluation Strategy

In the context of this thesis, a key evaluation criterion is the percentage of runtime deadlines met. As the primary optimisation target, this metric serves as a central measure of success and is thus closely monitored throughout the experiments. However, beyond simply tracking whether a deadline was met, more granular data is necessary, such as the magnitude of deadline violations.

Using the detailed request metrics provided by the load-testing client, valuable information can be gathered across multiple workload executions. This includes the average response time and how severely it deviated from the target deadline over time. Furthermore, the actual computation time of workloads is recorded separately which can be used to analyse overhead by comparing it to the total response time. This is particularly useful to identify increased scheduling delays due to resource exhaustion.

As previously described, deadline adherence is the primary optimisation target, however, efficient resource utilisation becomes increasingly important to achieve the specified SLOs, especially as the request-load grows. With resources within the cluster being limited, resource waste must be minimised while execution deadlines are still reliably met. Consequently, significant over-provisioning is not a sustainable approach, as it reduces the number of workloads that can be scheduled concurrently. This can lead to deadline violations for workloads that cannot be scheduled immediately due to resource exhaustion.

To evaluate the relation between resource efficiency and deadline adherence, different load scenarios are examined, showcasing how key metrics evolve over time as conditions change. This approach helps to determine how well the system handles stress and maintains runtime targets, even when operating under high-load conditions characterised with greater resource contention and scheduling delays.

Lastly, to put the obtained results into perspective and evaluate the developed approach, comparisons against commonly used resource allocation schemes by FaaS providers as well as the default `kube-scheduler` are performed. The specific baseline policies used for comparison are detailed in Section 6.1.

5. Implementation of DHRT

This chapter provides a comprehensive overview of the software solution developed in the scope of this thesis, which is subsequently referred to as **Deadline- and Heterogeneity-aware Resource Tuner (DHRT)**. To begin, a high-level overview of the system is presented, highlighting its main components and their responsibilities. Subsequent sections provide a more in-depth view of the individual components, discussing important implementation details, design choices and optimisation strategies.

5.1. System Overview

This section provides an overview of the architecture and functionality of DHRT, initially abstracting some of the finer details of the components and their interactions.

The primary objective of DHRT is to optimise and automate resource allocations and workload scheduling based on user-specified execution deadlines in heterogeneous clusters. Figure 5.1 illustrates the general architecture of the system, highlighting key resources and their interactions. Each component of DHRT has a distinct responsibility to achieve this goal:

- **Heterogeneous Deadline Scheduler (HDS):** Responsible for scheduling serverless workloads to suitable nodes, considering available node capabilities and workload requirements. The component is implemented in Go to natively integrate with the Kubernetes Scheduling Framework.

- **Inplace Resource Tuner (IRT):** Dynamically adjusts Resource Requests and Limits of executing workloads based on real-time metrics to prevent resource over- or under-provisioning and improve deadline adherence. To achieve this, Kubernetes' alpha-feature `InPlacePodVerticalScaling` and the Kubernetes' Informer library are utilised. It is also developed in Go due to its tight integration with Kubernetes' tools.

- **Heuristic Resource Estimator (HRE):** Analyses historical workload executions to determine good resource allocations (Requests and Limits) for each available node type using heuristics. Its objective is to ensure workloads meet execution deadlines while maximising resource efficiency. It is implemented in Python as its primary task is to analyse historical data and execute heuristic-based optimisation algorithms.

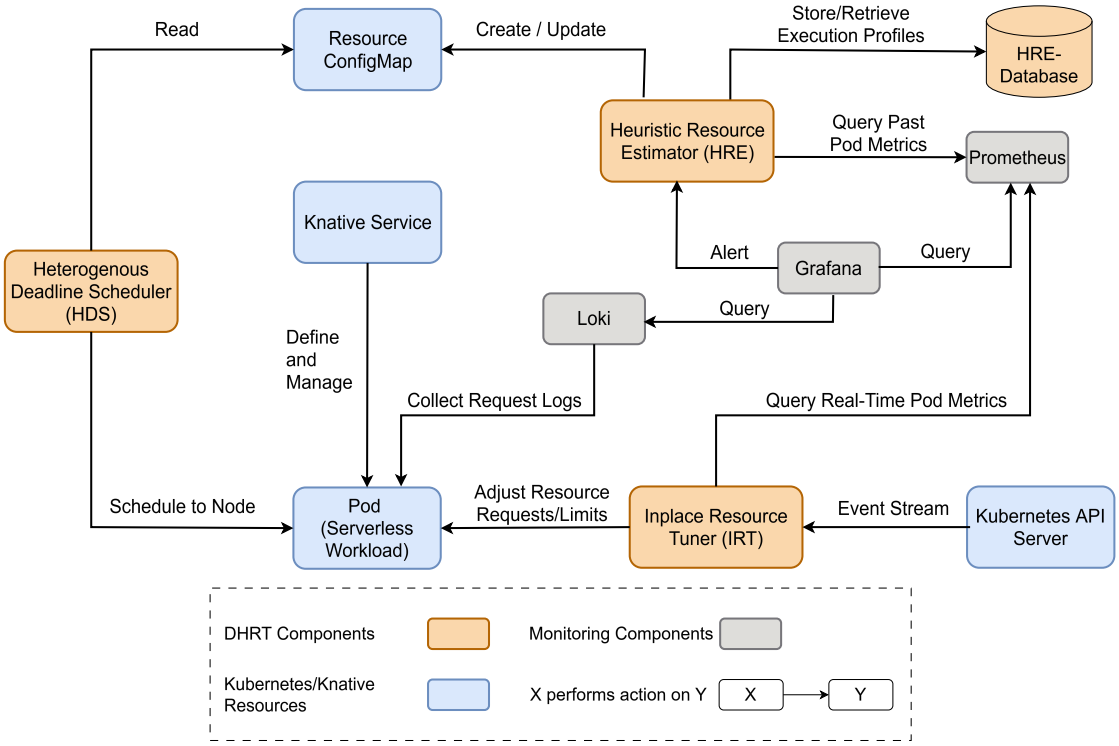

Figure 5.1.: **High-Level overview of System Components.** The diagram highlights the relations among the developed components, Kubernetes and Knative resources as well as the monitoring stack.

5.1.1. System Workflow

This section outlines the general workflow of DHRT, from the initial invocation of a serverless workload to its successful completion. Throughout this process, several important steps take place to optimise resource utilisation and workload performance. This includes the scheduling process, applying in-place Pod adjustments and the use of runtime metrics to iteratively refine scheduling decisions and resource allocation. The key steps of a single iteration are described below. Additionally, Figure 5.2 further outlines the system's workflow, highlighting the interaction of components throughout the lifecycle of a serverless workload.

1. **Workload Triggered.** A serverless workload is invoked via an HTTP request, triggering the scheduling process for the corresponding Pod.

2. **Pod Scheduling.** HDS retrieves the workload's resource estimates for each available node type from a ConfigMap to filter unviable nodes and rank the remaining ones to make a scheduling decision.

3. **In-Place Resource Adjustments.** During the execution of the workload, IRT periodically checks the Pod's resource utilisation metrics as it approaches its deadline,

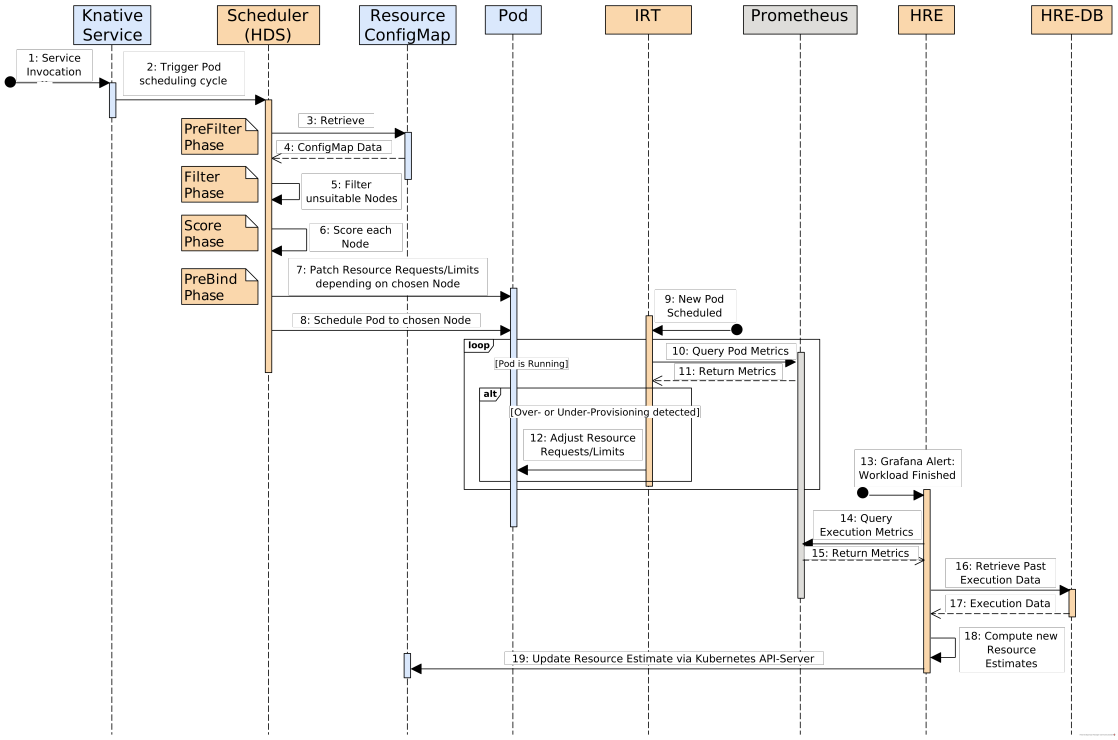

Figure 5.2.: **System Workflow and Component Interactions.** The Sequence-Diagram showcases the system's central workflow, representing one iteration of the feedback loop, encompassing a single workload execution. First, a workload is invoked and scheduled to a node by HDS. During its runtime, its resources are dynamically adjusted by IRT if there are signs of over- or under-provisioning. After the workload completes, HRE collects and analyses the Pod's resource utilisation metrics to update its resource estimates.

looking for signs of over- or under-provisioning. These dynamic adjustments become less frequent as information about the workload is collected and resource estimates are improved over time.

a. **Over-provisioning:** If CPU or memory usage is significantly below the requested amount, the respective Resource Request is decreased to free up capacity for other Pods and improve resource efficiency.

b. **Under-provisioning:** If CPU or memory usage nears its configured limit, causing the Pod to be throttled, it attempts to increase the affected Resource Request and Limit based on the leftover resources on the node. This vertical scaling takes effect within seconds, which helps to prevent performance degradation and ensure that runtime deadlines can be met. Naturally, such vertical scaling is only possible if the node has sufficient available resources to accommodate the increased Requests.

4. **Workload Completes.** Once the workload's execution finishes, Loki collects the request logs generated by the workload invocation. This information is queried by Grafana, which sends an alert to HRE containing the request metrics and workload metadata.

5. **Build Execution Profile.** HRE receives the alert and builds an execution profile for the invocation by querying resource metrics from Prometheus. This profile includes detailed resource usage metrics and runtime information, which is subsequently stored in a database, as shown in A.5.

6. **Update Resource Estimates.** Finally, HRE recomputes the recommended resource configuration by incorporating the newly gained information. The updated estimate is then stored for future invocations by updating the corresponding ConfigMap of the workload.

5.2. Heterogeneity-Aware Scheduling

The developed scheduler, subsequently referred to as HDS, is implemented using the Kubernetes Scheduling Framework, which is described in more detail in Section 2.2. A key objective of HDS is to actively integrate the available node heterogeneity into the scheduling process. To achieve this, several custom plugins have been developed, which are subsequently explained in more detail.

Overall, the scheduler's placement strategy tries to balance both the short-term efficiency and long-term cluster stability. In the short-term, it prioritises efficient resource utilisation based on the cluster's current state by densely packing nodes to avoid resource fragmentation and maintain high node utilisation. Over the long term, it tries to align workloads with complementary nodes, such as scheduling memory-intensive workloads to memory-optimised nodes. The goal is to prevent resource bottlenecks by avoiding significant resource imbalances on a node, such as cases where memory remains largely unallocated while CPU utilisation is very high. If these imbalances occur across many nodes, they can significantly reduce the scheduling flexibility for future workloads. Therefore, the challenge lies in finding a good balance, ensuring that short-term efficiency is not sacrificed for long-term stability and vice versa.

5.2.1. PreFilter Phase

The PreFilter stage generates a shared state for subsequent scheduling phases, which helps to avoid redundant computations and minimise accesses to the Kubernetes API Server. In the case of HDS, a ConfigMap generated by HRE is retrieved, containing the Pod's recommended resource allocations for each node type. An example of such a ConfigMap can be seen in Listing 5.2.

This step is necessary because Kubernetes' deployment model assumes a single resource

configuration per Pod, regardless of the node it is scheduled on. However, in heterogeneous clusters, this static resource model is often insufficient, as performance characteristics can vary significantly across nodes. This is particularly relevant for vCPU allocations, because different machine types may require varying amounts of vCPU to meet the same performance targets. By maintaining individual resource configurations, we can ensure that these variations are properly incorporated into the scheduling process.

Should no resource estimates be available because the workload has not executed previously on a specific node, then estimates from other node types are used as fallback configuration to guide the initial scheduling. While these may not fully account for machine-specific performance variations, they provide useful information about the workload's general resource demands. If no information is available at all, default values are utilised instead which try to cover common workload requirements (Requests: 1024 MiB, 1 vCPU; Limits: 2048 MiB, 2 vCPU).

Finally, the processed configurations are stored in the `cycleState`, which serves as a cache for the entire scheduling cycle of the Pod.

5.2.2. Filter Phase

The Filter phase is responsible for discarding any unviable nodes early in the scheduling process. The developed filter plugin works very similar to Kubernetes' `NodeResourcesFit` plugin, which checks whether a node has sufficient resources to accommodate the Pod.

The key distinction is that instead of relying on a single resource configuration across all nodes, it evaluates each node against the previously retrieved individual resource configurations. This adaptive approach essentially means that the filtering threshold can be different for each node.

5.2.3. Score Phase

The Score phase ranks each remaining node to evaluate its suitability for scheduling the Pod. After all scoring plugins have executed, an aggregated score between 0 and 100 is returned, with higher scores indicating a more favourable node for scheduling.

The fundamental strategy of HDS is to balance the workload-node compatibility with efficient cluster utilisation. Workloads are often best scheduled onto nodes that closely match their resource demands, such as placing memory-intensive workloads on high-memory nodes. However, in such a highly dynamic environment, static placement strategies alone are not enough. It is also necessary to account for real-time node utilisation levels to mitigate resource fragmentation.

To achieve this balance, the final score is based on a weighted aggregate of sub-scores, which are explained in more detail in the following sections. A brief overview of the

individual scores is listed below, ordered by their relative weight in the final score:

- **Resource-Alignment Score:** Prioritises nodes where the vCPU-to-memory ratio closely matches the determined workload's resource demands.
- **Resource-Balance Score:** Aims to reduce resource fragmentation by prioritising nodes where scheduling the Pod results in a more balanced distribution of CPU and memory utilisation.
- **Resource-Utilisation Score:** Evaluates CPU and memory availability, prioritising which are more densely packed.

Resource-Alignment Score

The objective of the Resource-Alignment Score is to prioritise nodes whose resource proportions align well with the workload's resource needs. This is particularly important in heterogeneous clusters, where mismatches between workload demands and node resources can lead to inefficiencies.

By aligning workloads with nodes that best fit their resource needs, it helps to prevent resource bottlenecks, for example caused by scheduling a compute-intensive workload on a memory-optimised node. Moreover, it can improve the overall performance by reducing resource contention and ensuring workloads run on nodes that can best support their execution characteristics.

As shown below, this score compares the workload's CPU-to-memory Request ratio to the node's resource ratio:

$$\text{ratio}_{\text{pod}} = \frac{\text{request}_{\text{cpu}}}{\text{request}_{\text{memory}}} \tag{5.1}$$

$$\text{ratio}_{\text{node}} = \frac{\text{capacity}_{\text{cpu}}}{\text{capacity}_{\text{memory}}} \tag{5.2}$$

After calculating the respective resource ratios, the Resource-Alignment Score ($S_{\text{alignment}}$) is computed in (5.3) as the absolute difference between the Pod's and the node's resource ratio. The result is then scaled by the maximum possible score of S_{max}.

$$\text{S}_{\text{alignment}} = (1 - |\text{ratio}_{\text{pod}} - \text{ratio}_{\text{node}}|) \cdot \text{S}_{\text{max}} \tag{5.3}$$

If $S_{\text{alignment}}$ is close or equal to S_{max}, it indicates that the workload's resource requirements closely follow the node's vCPU-to-memory ratio. In contrast, a low score suggests a mismatch between the Pod's resource demands and the node's capacity. However, this does not necessarily mean that the node is unsuitable for scheduling, rather, it prioritises nodes that are better aligned with the Pod's resource profile.

Lastly, as $S_{\text{alignment}}$ relies on workload data for a specific node, it serves a secondary purpose when no information is available yet. In this case, $S_{\max}$ is returned to encourage exploration early on and collect execution data across all available nodes. However, the impact of $S_{\text{alignment}}$ in the final node score is halved to prioritise other scoring criteria. This approach aims to prevent situations where the scheduler prematurely prioritises nodes where the workload has already executed.

Resource-Balance Score

The Resource-Balance Score (S_{balance}) evaluates how scheduling a Pod affects the node's vCPU-to-memory balance. The objective is to avoid resource fragmentation within a node, such as CPU exhaustion while memory remains underutilised. Such significant resource imbalances can limit the possibilities of future workload placements, which can lead to reduced resource efficiency.

S_{balance} measures whether placement of the Pod improves or worsens the resource balance on a node. To do this, a balance metric is defined in (5.4) to measure how evenly CPU and memory are utilised on a node. Next, the difference in balance before and after $(cpu', memory')$ scheduling is computed and scaled by $S_{\max}$, as shown in (5.5).

$$\text{balance}\big(\text{cpu}, \text{memory}\big) = 1 - |\text{utilisation}_{\text{cpu}} - \text{utilisation}_{\text{memory}}| \tag{5.4}$$

$$\Delta\text{balance} = \big(\text{balance}(\text{cpu}', \text{memory}') - \text{balance}(\text{cpu}, \text{memory})\big) \cdot S_{\max} \tag{5.5}$$

To add more significance to severe changes in the resource balance, the computed Δbalance is raised to the power of α (default = 1.20). As both improvements and degradations are possible, the sign must be retained, as shown in (5.6).

$$\Delta\text{balance}^* = \text{sgn}(\Delta\text{balance}) \cdot |\Delta\text{balance}|^{\alpha} \qquad \text{with } \Delta\text{balance}^* \in \left[-\frac{S_{\max}}{2}, \frac{S_{\max}}{2}\right] \tag{5.6}$$

Lastly, the transformed value (Δbalance*) is added to a base score corresponding to half of $S_{\max}$ to compute the Resource-Balance Score (S_{balance}). This ensures that no change results in a neutral baseline score, while negative and positive balance shifts are reflected with lower and higher scores respectively.

$$\text{S}_{\text{balance}} = \frac{\text{S}_{\max}}{2} + \Delta\text{balance}^* \tag{5.7}$$

While the previously described Resource-Alignment score prioritises resource compatibility, it is limited to static information. In contrast, the Resource-Balance score is computed based on current cluster conditions to dynamically promote healthy resource utilisation patterns across nodes.

Resource-Utilisation Score

The Resource-Utilisation Score simulates the scheduling of the Pod to the node and measures the impact on its resources. The updated resource utilisation (utilisation$'$), which represents the percentage of allocated CPU and memory respectively, is then compared to a defined target load-level. By default, the employed strategy works similar to a bin-packing approach, prioritising high utilisation nodes (close to 90% for both CPU and memory). This approach aims to improve short-term resource utilisation by densely packing nodes to reduce resource waste.

This score is computed separately for both CPU and memory, gradually penalising nodes as they deviate further from the target. The final score is then normalised by the maximum score, with higher scores indicating nodes whose utilisation is closer to the target.

$$\Delta\text{utilisation}_r = \left|\text{utilisation}'_\text{r} - \text{utilisation}_\text{target}\right|, \quad r \in \{\text{cpu}, \text{memory}\} \tag{5.8}$$

$$\text{deviation-factor}_r = 1 - \frac{\Delta\text{utilisation}_r}{\text{utilisation}_\text{target}} \tag{5.9}$$

$$\text{S}_\text{r} = \text{deviation-factor}_r \cdot \text{S}_\text{max} \tag{5.10}$$

By tuning the target utilisation via environment variables, different placement strategies can be realised, for example prioritising the least allocated node instead to achieve a more distributed resource utilisation across the cluster.

Final Weighted Node-Score

Finally, the individual scores described earlier are aggregated into an overall node score (S_node, with the highest-scoring node then selected for scheduling the Pod. By default, the following weights are used to compute the final weighted node score:

$$w_\text{alignment} = 2.0, \quad w_\text{balance} = 1.5, \quad w_\text{cpu} = 1, \quad w_\text{memory} = 1 \tag{5.11}$$

$$\text{S}_\text{node} = \frac{(w_\text{alignment} \cdot \text{S}_\text{alignment}) + (w_\text{balance} \cdot \text{S}_\text{balance}) + (w_\text{cpu} \cdot \text{S}_\text{cpu}) + (w_\text{memory} \cdot \text{S}_\text{memory})}{w_\text{alignment} + w_\text{balance} + w_\text{cpu} + w_\text{memory}} \tag{5.12}$$

Overall, this approach aims to balance current cluster conditions with static node characteristics, ensuring that Pods are placed on the most suitable node based on both real-time resource availability and resource compatibility between workloads and nodes.

Penalising Unsuitable Nodes

Lastly, if a workload repeatedly fails to meet its deadline on a specific type of node, a penalty is applied to the final score based on the severity of past deadline violations. However, this penalty is primarily reserved for highly specialised and computation-heavy workloads, where even a high allocation of vCPUs may not be sufficient on general-purpose nodes. Conversely, it can be applied to strictly sequential workloads which might require a compute-optimised node as they do not benefit well from increased vCPU allocations. By penalising such nodes, the scheduler strongly discourages placements where SLOs cannot be reliably maintained.

The computation of this penalty can be viewed in (5.13). Essentially, it reduces the node score based on the deadline capability, which represents the ratio of the best recorded execution time to the deadline. For example, a value of 1.50 indicates that the workload only completed within 1.5× its deadline. Higher values result in a stronger penalty, controlled by the exponential decay factor k (default = 3).

$$S_{\text{node}} = S_{\text{node}} \cdot \text{deadlineCapability}^{-k} \tag{5.13}$$

While this could also be used as a filter condition, applying a scoring penalty instead allows for more flexible scheduling decisions, as unsuitable nodes are only deprioritised rather than outright excluded. This keeps additional placement options available during high-load scenarios where a better placement might not be possible.

5.2.4. PreBind Phase

Once a node has been selected, additional operations can be performed in the PreBind phase before the Pod is finally bound to the node. This phase is used to address the previously mentioned restriction of Kubernetes, where Pod's have a fixed resource configuration, regardless of the node they are ultimately scheduled on.

Therefore, the PreBind phase dynamically adjusts the Pod's resource allocation based on the selected node, using the estimates previously retrieved in the PreFilter phase. This is done by directly patching the PodSpec using the `InPlacePodVerticalScaling` feature, ensuring that the the workload is provisioned with the appropriate resources based on the machine type.

5.2.5. Deployment and Usage of HDS

Since a custom scheduler operates within the control plane and influences core scheduling decisions, its deployment requires additional considerations beyond typical workloads. First, it requires the necessary permissions to perform scheduling operations, which involves tuning ClusterRoles and permission management. Additionally, the developed plugins must be registered at the respective phases of the scheduling cycle. To do so, a `KubeSchedulerConfiguration` must be created and passed to the scheduler as a startup

argument. The configuration of HDS is shown in Listing A.1 as an example.

The configured scheduler profile also disables some default plugins, namely during the Score and Filter phase. This is because HDS has a distinct approach to resource allocation. By default, Kubernetes assigns a fixed resource configuration to Pods at deployment time according to their manifest, meaning all Pod replicas have the same Resource Requests and Limits, regardless of the node they are scheduled on. HDS lifts this restriction by dynamically adjusting resource configurations based on the selected node. Therefore, to prevent conflicts with Kubernetes' static resource model, some default scheduling plugins were disabled.

Lastly, to improve portability and ensure that HDS can run in any Kubernetes cluster regardless of control plane permissions, it is deployed as a secondary scheduler alongside the default `kube-scheduler`. As a result, it is only responsible for scheduling a subset of Pods. To use HDS, deployments must set the `spec.schedulerName` property in their manifest. For serverless workloads, this requires enabling the `kubernetes.podspec-schedulername` feature in Knative.

5.3. In-place Resource Tuning of Pods

Implemented by the IRT component, in-place resource tuning is a critical feature of this system, enabling real-time updates to a Pod's Resource Requests and Limits while it is still executing.

One one hand, this functionality helps to correct bad initial resource allocations which are common in online optimisation settings. Underestimating resource needs can lead to missed deadlines or even failed executions, while overestimating can result in inefficient resource utilisation, as resources are reserved but not required. Alternatively, not specifying any Resource Requests or Limits is discouraged, as it not only complicates scheduling and capacity planning, but can cause performance degradation due to resource contention among Pods.

Therefore, without in-place adjustments, we would either need to perform dedicated workload profiling in advance, contradicting the agile nature of the serverless deployment model or alternatively, accept a high risk of missed deadlines or inefficient resource usage during the initial workload executions.

On the other hand, by dynamically adapting allocated resources according to real-time metrics, IRT can help mitigate performance variations and prevent deadline violations. For instance, as a workload approaches its deadline, the vCPU allocation can be increased to help maintain a timely completion, similar to the approach taken by [28, 29].

The following sections describe the adaptive resource tuning strategy in more detail.

5.3.1. Tracking Resource Updates with Kubernetes Informer

In principle, IRT is implemented using the Kubernetes Informer API, which allows it to monitor changes to any Kubernetes resource, caching updates as they arrive. This significantly reduces the load on the Kubernetes API-Server, as it reduces the need for constant polling. Specifically, IRT tracks updates from three primary resources:

- **Nodes**: Keep track of available nodes and the Pods that are currently scheduled there to retrieve the currently remaining resource capacity.
- **Pods**: Continuously monitor deployed workloads while accessing Pod metadata, such as the current resource allocation.
- **ConfigMaps**: Maintain the most up-to-date resource estimate provided by HRE.

5.3.2. Timing and Frequency of In-Place Pod Resizing

IRT detects new workload invocations by monitoring updates to Pods, namely when the `spec.nodeName` field is set, which indicates that the Pod has just been scheduled to a node. At this point, IRT schedules its first resource evaluation of the workload.

The timing and frequency of these evaluations depends significantly on the amount of information available, as shown in Table 5.1. In principle, evaluations are scheduled at specific checkpoints that correspond to fractions of the workload's specified deadline. However to ensure that reliable metrics are available and to allow previous resize actions to take effect, each iteration requires a minimum time to have passed, which is by default set to 30 seconds.

As more execution data is gathered, DHRT can increasingly rely on the resource estimates of HRE, reducing the need for real-time corrections. This makes resource allocations more predictable over time, improving cluster stability and enabling more informed scheduling decisions. At this point, resource adjustments are primarily used a precautionary measure when the workload closely approaches its deadline.

5.3.3. Pod Resource Evaluation

Before resizing a Pod's resource allocation, it must first be determined whether an adjustment is necessary in the first place. To do so, a snapshot of the Pod's current state is generated by retrieving CPU and memory utilisation metrics from Prometheus. Each resource is then evaluated against predefined thresholds as showcased in Listing 5.1.

If the observed usage exceeds 95% of its defined Resource Limit, indicating a risk of throttling, the system triggers a scale-up operation for that resource. Conversely, if the utilisation falls below 70% of the resource Request, a scale-down operation is performed to free-up idle resources.

Description	Recorded Workload Executions	Evaluation Checkpoints (% of Deadline)
Initial Phase (no data available)	0	20, 40, 60, 80
Early Learning Phase (limited data)	≤ 3	40, 60, 80
Stable Phase (moderate data)	≤ 5	60, 80
Mature Phase (sufficient data)	> 5	80

Table 5.1.: **Frequency of In-Place Pod Resizing**. The table outlines how the frequency of Pod evaluations is determined according to different phases of data availability, starting from the initial phase with no prior execution data on a node pool to the mature phase with sufficient data for more accurate predictions.

This iterative process ensures that each resource is adjusted in real-time according to the workload's demands, maintaining both performance and resource efficiency.

```
// Loop through each resource (CPU, Memory)
for _, rs := range podSnapshot.ListMetrics() {
  // Check if resource is at risk of throttling (within 95% of limit)
  if rs.resourceUsage >= (rs.resourceLimit * scaleUpThreshold) {
    // Scale up resources for this resource type
  }
  // Check if resource usage is below idle threshold (50% of request)
  else if rs.resourceUsage <= (rs.resourceRequest * scaleDownThreshold) {
    // Scale down resources for this resource type
  }
}
```

Listing 5.1: **Pod Resizing Logic.** Both CPU and Memory utilisation of the Pod are checked for signs of throttling or idle resources. The default value for the `scaleUpThreshold` is set to 0.95, whereas `scaleDownThreshold` is set to 0.70.

5.3.4. Vertical Scaling Strategy

Once it is decided that a resource resize is necessary, the next step is to determine the magnitude of the adjustment. Striking the right balance is critical, as changes that are too aggressive can lead to instability and resource overcommitment, while insufficient updates may fail to respond in time and have a meaningful impact. Different approaches are taken for both upscaling and downscaling, which are now described in more detail.

Upscaling Algorithm

A key factor to consider for upscaling is the current node utilisation. If the node is already operating at or near full capacity, increasing the resource allocation of a Pod could lead to overcommitment. Conversely, if the node has a large amount of idle resources, IRT

can be more aggressive with upscaling and allocate more resources.

To compute the new Resource Request, a gradual scaling approach is applied, where upscaling becomes less aggressive as node utilisation increases. The precise adjustments in relation to the current utilisation are illustrated in Figure 5.3. Essentially, the scaling strategy is divided into three levels:

- **Low Utilisation ($<$ 30%):** The maximum increase of 1.25× is applied to make efficient use of idle resources.
- **Moderate Utilisation (30% - 90%):** The scaling factor gradually decreases from 1.25× to 1.05×.
- **High Utilisation ($>$ 90%):** No scaling is applied to avoid overloading the node and risking performance degradation.

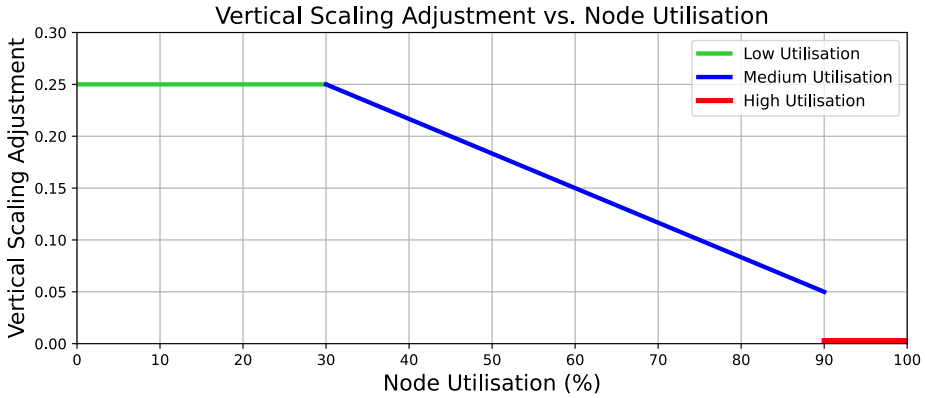

Figure 5.3.: **IRT Upscaling Magnitude.** The plot showcases the default behaviour to determine the upscaling magnitude. The vertical scaling factor gradually decreases as the node utilisation increases. By default, no scaling beyond 90% utilisation is performed.

Lastly, to maintain stable performance, no scaling beyond 90% of the node's capacity is performed. However, this can be temporarily relaxed to 95% capacity if the workload is almost near its deadline. This is done to prioritise deadline adherence, as the resource impact is only short-term as the workload is expected to complete soon.

Downscaling Algorithm

On the other hand, downscaling is primarily dependent on the difference between the requested resources and the actual resource consumption, in other words, how many reserved resources of the Pod are effectively idle:

$$R_{\text{idle}} = R_{\text{request}} - R_{\text{usage}} \tag{5.14}$$

Subsequently, to ensure that downscaling is gradual and does not result in under-provisioning, a scaling factor is computed based on the amount of idle resources. This scaling factor is defined as:

$$\text{scale-down-factor} = 1 - \min\left(\frac{R_{\text{idle}}}{R_{\text{request}}}, 0.5\right) \tag{5.15}$$

The scaling factor limits the proportion of idle resources that can be released, preventing too severe reductions with a maximum factor of 0.5. The updated Resource Request (Request$'$ is then computed by taking the current usage value and incrementing it by a portion of the idle resources, as determined by the previously computed scale-down factor.

$$R'_{\text{request}} = R_{\text{usage}} + R_{\text{idle}} \cdot \text{scale-down-factor} \tag{5.16}$$

5.3.5. Reactionary Scaling for Long-Pending Pods

When dealing with strict runtime deadlines, scheduling delays as a result of resource shortages can significantly impact SLOs. To counteract excessive pending times during periods of high load, IRT dynamically increases the vCPU allocation of Pods that have been in a pending state for a notable amount of time (5-10% of deadline). This measure aims to offset the current delay by accelerating execution once the Pod is finally scheduled.

This CPU boost is applied adaptively based on the severity of the scheduling delay relative to its deadline. If enough resources are left available on the node, this temporary adjustment can allow workloads to recover from long wait times and still complete within their deadline.

$$\text{boost-factor} = 1 + \frac{t_{\text{pending}}}{t_{\text{deadline}} - t_{\text{pending}}} \tag{5.17}$$

5.3.6. Maintaining Up-To-Date Resource Estimates

Since IRT frequently applies temporary resource adjustments, their impact on resource efficiency must be considered when multiple subsequent workloads run on the same Pod. In our deployment model, Knative is configured with a concurrency value of 1, ensuring that only a single workload executes on a Pod at a time. However, if a new request of the same workload arrives before a scale-down is triggered, it may be scheduled on the same Pod. In this case, the previous temporary adjustments would persist throughout the execution of the new workload.

To handle such scenarios, IRT periodically reverts any temporary resizes and updates the Pod's resource configuration based on the most recent estimate provided by HRE. This ensures that short-term boosts of a previous workload do not persist for the whole duration of the next invocation. Additionally, it keeps the resource configuration up-to-date even for long-running Pods.

5.3.7. Performing In-place Pod Resource Updates

To apply Pod resource updates, Kubernetes' alpha feature `InPlacePodVerticalScaling` is used, which is described in more detail in Section 2.4.6. It allows to update Resource Requests and Limits of a Pod during its execution without requiring a restart. By issuing a JSON-merge patch through the Kubernetes API, the Pods' manifest can be modified with the updated values.

An example of how this can be done via `kubectl` is shown in Listing A.3, whereas Listing A.4 presents the implementation in IRT using the Kubernetes client library.

5.4. Data Processing and Heuristic-based Resource Estimation

A key requirement of DHRT is the continuous collection and analysis of workload execution data. This is done by HRE, which collects and processes valuable resource metrics of workloads to build execution profiles after each invocation. These profiles are used to refine resource estimates to ensure that future workload invocations reliably meet their deadline. At the same time, these resource estimates provide important input for scheduling decisions made by HDS. The exact role and functionality of HRE is explored in more detail in the upcoming sections.

5.4.1. Webhook-Driven Workflow

A primary challenge of this component was to obtain precise information about the execution time of workloads. Therefore, we cannot rely on Pod metrics alone, but rather require accurate and reliable timing data of individual requests to identify the start and end of workload executions.

To achieve this without altering the workloads themselves, Knative's Queue-Proxy sidecar container, responsible for managing traffic to the user container, was utilised. While Queue-Proxy natively captures aggregate metrics, more granular data was needed to fit the system's needs. Therefore, the `request-log-template` in Knative's observability configuration was updated, as demonstrated in Listing A.2. These logs capture essential details, including service metadata and request-specific data such as status and latency. Based on this information, we can precisely identify workload executions, including the Pod and Node on which they ran, as well as their exact response time.

Nevertheless, these logs are not widely accessible or readily available for querying, especially in a serverless environment where Pods are ephemeral. Therefore, an efficient approach for collecting these logs and integrate them into the monitoring stack is necessary. To solve this, Loki [48] is utilised to periodically scrape and store these request logs.

Lastly, we need a way to notify HRE whenever new data is available to realise the central feedback loop and refine resource estimates. For this purpose, Grafana Alerting [49] is used to create an alert whenever this specific log entry, generated by Queue-Proxy, is detected. This alert is then sent to HRE via a webhook. The specific details of the alerting configuration are described in Section 5.5.3.

5.4.2. Data Retrieval and Aggregation

Whenever the previously described request alert is received, HRE begins to retrieve and aggregate data from multiple sources to build an execution profile. Relevant metadata is extracted directly from the received alert, which is then used to query Prometheus to retrieve resource utilisation metrics throughout the workload's execution. The returned raw time-series data is then aggregated for further processing. An example of a generated execution profile of a workload can be viewed in the Listing A.5.

CPU Metrics Profile

To fetch the CPU utilisation data, a range-query is performed using the precise timing data extracted from the received alert to determine the workload's start- and end-time. This query, as demonstrated below with template values, returns the total vCPU usage rate (in cores) for the specified Pod over a sliding 40-second window.

This process is repeated for the entire time range specified by the `start` and `end` parameters. To ensure consistent behaviour across different container runtimes, container and image filters are used to exclude pause containers and Pod-level metrics

```
sum(rate(container_cpu_usage_seconds_total{image!="",
container_name!="POD", namespace="{namespace}", pod="{pod_name}"}[40s]))
```

The returned time-series data is then aggregated to generate a CPU usage snapshot, providing the minimum, maximum, and average usage throughout the Pod's execution. These metrics offer valuable insights into resource consumption trends during the workload execution and are more practical to integrate into subsequent resource optimisation computations. Furthermore, the CPU Request and Limit are tracked over time, enabling the comparison of the Pod's actual usage against its allocated resources. This provides a clearer context to make more accurate estimates for future executions.

Memory Metrics Profile

The memory profile is constructed similarly, using a range-query to retrieve the memory usage in bytes throughout multiple points in the workload's execution:

```
sum(container_memory_working_set_bytes{image!=", container_name!="POD",
              namespace="{namespace}", pod="{pod_name}"})
```

Unlike the CPU profile, memory metrics are aggregated using the 95th percentile and the maximum observed usage. This approach was chosen because, unlike CPU throttling, running out of memory is typically worse, as it can result in an Out of Memory (OOM) termination of the application. Therefore, we need to find reliable memory estimates that are sufficient throughout the whole workload execution.

5.4.3. Optimisation of Resource Configurations

This section describes how the previously generated historical execution data is integrated into the resource optimisation process. The objective is to find a resource configuration (Requests and Limits) which allows a workload to reliably complete within its runtime deadline. In the case of heterogenous clusters, these configurations can vary between nodes to account for different performance capabilities.

At the same time, it is essential to maintain efficient resource utilisation throughout the cluster, which means resources must be allocated carefully. As such, the optimisation algorithm targets configurations that can finish at around 90% of the deadline, subsequently referred to as `soft-deadline`. This threshold was selected to avoid deadline violations due to slight performance variations or typical overhead such as container startup and network latency.

Heuristic-Based CPU Estimation

To estimate the CPU requirements of a workload, weights are assigned to each execution profile depending on how close its runtime is to the soft-deadline target. These weights are computed using an exponential decay function, which progressively assigns lower weights to profiles that deviate further from the target runtime.

The runtime deviation from the soft-deadline is first computed as shown in (5.18). Then, a weight w_i for each profile i is computed in (5.19), where k represents the decay parameter that controls how fast the weight decreases as the runtime deviates further from the target. By default, k is set to 0.20.

$$\text{deviation}_i = \text{soft-deadline} - \text{observed-runtime}_i \tag{5.18}$$

$$w_i = \exp\left(-k \cdot |\text{deviation}_i|\right) \tag{5.19}$$

This approach ensures that the best-performing runs, in other words those closest to the runtime target, have a stronger influence on the final CPU estimate. However, currently the algorithm does not penalise deadline violations, as it assigns the same weight to both deadline violations and early terminations if they have the same absolute deviation from the target runtime. To account for this, a penalty is introduced to decreased the weight

by 70% for hard-deadline violations and by 20% for soft-deadline violations.

$$w_i = \begin{cases} w_i \cdot 0.3, & \text{if hard-deadline violation} \\ w_i \cdot 0.8, & \text{if soft-deadline violation} \\ w_i, & \text{otherwise} \end{cases} \tag{5.20}$$

Once the weights are computed, a weighted average of the average CPU utilisation from each execution profile i (where $i = 1, \ldots, n$) is calculated in (5.21). The result is used to update the workload's CPU Request for future invocations. The CPU Limit is then set slightly higher, typically 5-10% above the Request.

$$\text{cpu}_{\text{request}} = \frac{\sum_{i=1}^{n} w_i \cdot \text{avg-cpu-usage}_i}{\sum_{i=1}^{n} w_i} \tag{5.21}$$

Lastly, to guide the algorithm towards discovering good solutions and improve convergence rate, a steering factor is applied. Without this, estimates would only take the average utilisation into account, lacking awareness of SLOs and relying primarily on resource resizing of IRT to discover good configurations.

Therefore, the computed $\text{cpu}_{\text{request}}$ is scaled relative to how close current execution times are to the targeted soft-deadline. To approximate the quality of the current estimate, the `deadline-ratio` of each profile (as computed in (5.22)) is used, which is calculated as the fraction of execution time to the deadline. Using the weights previously computed in (5.19), a `weighted-deadline-ratio` is computed in (5.23) which serves as a quality metric for the computed CPU estimate. The idea behind this approach is that if the `weighted-deadline-ratio` significantly deviates from the targeted soft-deadline, as shown in (5.24), then the computed CPU request is likely to be inaccurate.

$$\text{deadline-ratio}_i = \frac{\text{observed-execution-time}_i}{\text{deadline}} \tag{5.22}$$

$$\text{weighted-deadline-ratio} = \frac{\sum_{i=1}^{n} w_i \cdot \text{deadline-ratio}_i}{\sum_{i=1}^{n} w_i} \tag{5.23}$$

$$\Delta\text{deadline-ratio} = \text{soft-deadline-ratio} - \text{weighted-deadline-ratio} \tag{5.24}$$

The computation of the steering factor based on Δdeadline-ratio is shown in (5.25). By default, the magnitude of downscaling is smaller than for upscaling to prevent excessively reducing vCPU allocation and risk introducing deadline violations. Therefore, their default values are $k_{\text{downscale}} = 0.5$ and $k_{\text{upscale}} = 0.75$.

$$\text{steering-factor} = \begin{cases} 1 - k_{\text{downscale}} \cdot \Delta\text{deadline-ratio}, & \text{if } \Delta\text{deadline-ratio} > 0 \\ 1 + k_{\text{upscale}} \cdot (-\Delta\text{deadline-ratio}), & \text{if } \Delta\text{deadline-ratio} < 0 \\ 1, & \text{otherwise} \end{cases} \tag{5.25}$$

Lastly, the previously computed estimated for the CPU Request in (5.21) is scaled based on the steering factor:

$$\text{cpu}'_{\text{request}} = \text{cpu}_{\text{request}} \cdot \text{steering-factor} \tag{5.26}$$

To see an example computation of the CPU Request estimate, please refer to Section A.1.5.

Heuristic-Based Memory Estimation

Similar to the CPU optimisation strategy, past execution profiles are used to determine memory Requests and Limits. While vCPU allocations typically have a larger impact in meeting runtime deadlines, for memory we need to ensure sufficient allocation throughout the workload's execution to avoid OOM errors.

To compute a memory estimate, a weighted average is applied, with weights reflecting a recency factor that slightly prioritises more recent execution runs. This prioritisation is rather moderate, with the most recent run receiving a additional weight of 25%, gradually decreasing to 5% for the fifth most recent run as shown in (5.27). This can help to integrate new information more quickly and account for potential short-term variability.

$$w_i = \begin{cases} 1.25 - 0.05(i-1), & \text{if } 1 \leq i \leq 5 \\ 1, & \text{if } i > 5 \end{cases} \tag{5.27}$$

Finally, the memory Request is computed as the weighted average of the 95th percentile memory usage of each execution profile i (where $i = 1, \ldots, n$). This estimate should ensure that the workload has enough memory throughout its execution. On the other hand, the memory Limit is set to the weighted maximum observed usage across executions runs, with an added buffer to ensure additional headroom for potential spikes.

$$\text{memory}_{\text{request}} = \frac{\sum_{i=1}^{n} w_i \cdot \text{usage-p95}_i}{\sum_{i=1}^{n} w_i} \tag{5.28}$$

$$\text{memory}_{limit} = \frac{\sum_{i=1}^{n} w_i \cdot \text{usage-max}_i}{\sum_{i=1}^{n} w_i} \cdot \text{memory-buffer} \tag{5.29}$$

5.4.4. Integrating Node Pools for Resource Estimation

Clusters often contain multiple instances of the same node type, often referred to as a node pool. Since HRE makes node-specific resource estimates, it is beneficial to aggregate execution data from nodes within the same pool, as they are expected to have the same performance characteristics. Therefore, by aggregating the execution data of identical machine types, good resource allocations can be found without requiring a high number of workload executions on each individual node.

To achieve this, HRE assigns nodes to groups based on a configurable node label specified via an environment variable. These labels are often assigned automatically during cluster

creation. For example, GKE uses the label `cloud.google.com/gke-nodepool` to group its nodes by default.

However, it should be noted that this feature should not be used if there is a significant performance difference between identical nodes. As DHRT is deployed on virtual hardware, the lack of information regarding the underlying hardware can be problematic. If performance within a node pool drastically varies, then resource estimates will fluctuate depending on scheduling decisions and cannot be accurately identified.

This performance variability can occur because cloud providers often mix different processor platforms according to availability, even within the same machine type. For example, Google Cloud's [26] N1 nodes may be based on Intel Sandy Bridge, Ivy Bridge, Broadwell, Haswell and Skylake CPU platforms. To try and mitigate this uncertainty, a minimum CPU platform can be specified when creating a compute cluster [50].

5.4.5. Updating a Pod's Resource Profile

Once the resource estimates have been recomputed to incorporate the latest execution data, they must be made available for other components. However, because Pods may have different resource requirements to meet their deadlines based on the type of node they are deployed on, simply updating the Resource Requests and Limits of the corresponding Knative Service is not sufficient. This is because Kubernetes allows only a single configuration to be defined at deployment time, making per-node adjustments impossible through direct updates.

Therefore, to achieve this behaviour while keeping individual components of DHRT decoupled, the updated resource information is instead stored in a ConfigMap. This information is then used to dynamically update the Pod resource specification as the workload is scheduled to a specific node.

An example of such a configuration is provided in Listing 5.2, which contains an entry with the recommend resource allocation and important metadata for each node pool. This includes the number of previously observed executions within a node pool, as well as the deadline capability, which represents the best achieved execution time relative to the deadline.

```
apiVersion: v1
data:
  node-pool-c3: |-
     {
        "requests": {"cpu": "250m", "memory": "256Mi"},
        "limits": {"cpu": "275m", "memory": "290Mi"},
        "recorded_runs": 6,
        "deadline_capability": 0.81,
     }
```

```
10   node-pool-n1: |-
11     {
12       "requests": {"cpu": "400m", "memory": "256Mi"},
13       "limits": {"cpu": "440m", "memory": "290Mi"},
14       "recorded_runs": 4,
15       "deadline_capability": 0.89,
16     }
17 kind: ConfigMap
18 # ...
```

Listing 5.2: **Example ConfigMap of a Pod's Resource Estimates.** The example ConfigMap contains information about the recommended resource configuration of a workload for two different node types.

5.4.6. Interplay with IRT

The components HRE and IRT are an important part of the resource management process, which are responsible for iteratively improving a workload's resource configuration and adaptive vertical scaling respectively. As such, these two components are inherently coupled and affected by the actions of each other. Since HRE relies on historical execution data, any modifications introduced by IRT during the execution inherently affect subsequent resource allocation estimates.

Particularly, IRT helps to discover good resource allocations by proactively scaling vCPU allocations as workloads approaches their deadline. With this adaptive approach, resource needs can be found much quicker, allowing HRE to converge to good estimates more effectively. However, this dependency must be carefully balanced. If resource estimates start to implicitly depend on frequent dynamic adjustments, such as CPU boosts, it can lead to deadline violations if the expected adjustments cannot be applied due to resource constraints. To mitigate this, the system gradually shifts from this reactive model to a more stable one, relying less on real-time resource scaling and more on historical data instead.

5.5. Monitoring and Observability

A key requirement of DHRT is the availability of reliable and insightful metrics. Consequently, closely monitoring the execution of workloads is essential, not only to react in real-time but also to analyse and learn from resource utilisation patterns of past executions. This data is a key part to evaluate and improve scheduling and resource allocation decisions within DHRT, as it continuously feeds back into the system to refine decisions for future workload invocations.

Therefore, a robust and scalable monitoring solution is required to collect, store and query the large amount of metrics emitted in Kubernetes clusters. The following section will

describe how widespread monitoring and observability solutions are integrated and used to address these challenges.

5.5.1. Metric Collection using Prometheus

Prometheus [51], is an open-source monitoring solution and time-series datastore that has become the *de facto* standard for Kubernetes environments. By defining scrape-jobs, Prometheus periodically collects metrics of configured endpoints and stores the received data as time-series values. This can include a variety of metric sources at different granularity levels, ranging from cluster-wide metrics to individual container-level metrics and even custom application metrics.

For our use-case, an essential metric target is cAdvisor (Container Advisor) [52], which is natively embedded into the kubelet node agent. It provides detailed resource utilisation metrics for all containers running within Pods.

Given the importance of these metrics to guide scheduling and resource allocation decisions, the scrape interval for cAdvisor was reduced from the default 30s to 10s, as shown in the configuration in Listing 5.3. The goal of this reduced interval is to collect more fine-grained metrics to improve the responsiveness and accuracy of resource optimisation decisions.

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: prometheus-kube-prometheus-kubelet
  namespace: monitoring
spec:
  endpoints:
  - path: /metrics/cadvisor
    port: https-metrics
    interval: 10s  # Updated interval for cAdvisor metrics
  # ... other configuration details omitted
```

Listing 5.3: **Prometheus kubelet Service Monitor Configuration.** To configure Prometheus to scrape cAdvisor's metric endpoint more frequently, the interval was updated to 10s in the corresponding ServiceMonitor.

5.5.2. Log Aggregation - Loki

Loki [48] is a highly scalable log-aggregation system which integrates natively with Prometheus and Grafana. Similar to how Prometheus scrapes emitted metrics, Loki scrapes and collects log entries from active containers. This is particularly useful in serverless environments, where Pods are ephemeral, which makes logs short-lived and difficult to trace. In our case, important request details are logged by Knative's Queue-Proxy after the execution of serverless workloads, which serve as an important data source for our

analytics component HRE. An example of such a log is shown in Listing 5.4.

Loki not only aggregates these logs, but also indexes key metadata from the Pod, including important information such as the node it is deployed on. This unified view makes it very efficient to trace and analyse individual workload executions.

```
{
  "requestMethod": "POST",
  "requestUrl": "/",
  "status": 200,
  "latency": "89.477208897s",
  "protocol": "HTTP/1.1",
  "revisionName": "power-iteration-moderate-00001",
  "Namespace": "workloads",
  "Service": "power-iteration-moderate",
  "Configuration": "power-iteration-moderate",
  "PodName": "power-iteration-8c8d68987-vlsgk",
  "PodIP": "10.244.0.8"
}
```

Listing 5.4: **Example Request Log Data.** The showcased log is emitted by Knative's Queue-Proxy after each workload invocation, containing important metadata as well as request metrics.

5.5.3. Observability - Grafana

Grafana [49], an open-source monitoring and analytics solution, serves as a central hub to manage and visualise the large amounts of data collected by Prometheus and Loki. To make this data more accessible and easier to interpret, Grafana offers powerful tools to transform raw time-series data into insightful graphs and charts using custom dashboards. This improved observability helps to make metrics more accessible to closely monitor key components within a cluster

Moreover, Grafana provides powerful alerting capabilities based on query results, which we utilise to notify HRE of workload terminations. A snippet of this alert configuration is shown in Listing 5.5. This alert expression triggers as soon as the specified request log of Queue-Proxy, as shown in 5.4, is generated and collected by Loki. Once triggered, the alert is sent to HRE via a webhook, which then processes the received data to build an execution profile of the corresponding workload. This profile, as seen in A.5, stores resource metrics and metadata of an individual workload execution. Lastly, to make sure that the most recent metrics are considered, a `group_wait` time of 10 seconds is configured to introduce a slight delay before the alert is sent.

```yaml
datasource:
  type: loki
expr: |
  (count_over_time(
    (
      {namespace="workloads", container="queue-proxy"}
      | json
      | stream="stdout"
      | json
    )[1m]
  ) > 0)
# ...
notification_settings:
  receiver: hre
  group_by:
    - "..." # do not group alerts
  group_wait: 10s # wait 10s before sending alert
```

Listing 5.5: **Grafana Alert Configuration for Queue Proxy Logs.** This example shows a snippet of the utilised Grafana alert configuration that triggers a webhook notification to HRE when the defined expression evaluates to true. To ensure that each log creates its own alert, alert-grouping has been disabled.

5.6. Deterministic Pod Termination in Knative Serving

During initial testing, an issue was found that affects long-running workloads with a concurrency setting of 1. With this configuration, the KPA updates the number of Pods according to the number of currently active requests. While there are no issues during upscaling, problems can occur when the number of concurrent requests drops. In this case, the KPA updates the ReplicaSet to decrease the number of Pod replicas. However, since Kubernetes lacks insight into which Pod is idle and which one is actively processing a workload, it often terminates the wrong Pod and forcefully stops an ongoing execution.

To mitigate this and make Pod deletion deterministic, a minor fork of Knative Serving was created to address this issue. It uses Kubernetes' beta feature `Pod-Deletion-Cost` [53] to prioritise retention of Pods with active in-flight requests. The implementation uses the Queue-Proxy sidecar to dynamically adjust the Pod-Deletion-Cost property, increasing it when a request is in progress and subsequently decreasing it once the request has completed. This adjustment improves the behaviour for long-running serverless workloads by preventing arbitrary Pod deletions.

5.7. System Deployment and Usage

DHRT can be deployed on any Kubernetes cluster running version 1.27 or higher, with the `InPlacePodVerticalScaling` feature gate enabled. The following section briefly

showcases how DHRT can be used with an exemplary workload deployment.

Detailed deployment instructions are provided in the GitLab repository, as described in A.1. The provided setup covers the cluster creation and the deployment of all system components, including the monitoring stack and Knative.

5.7.1. Workload Deployment

To make use of DHRT, developers can deploy serverless workloads with minimal configuration overhead. DHRT then automatically manages the resource allocation and scheduling of the workload to meet the specified SLO, without requiring further user intervention. A key design objective was to keep the deployment process as simple as possible to maintain the simplicity of serverless computing.

Workloads are deployed as a standard Knative Service with a few additional properties to indicate that the workload should be managed by DHRT. An example of such a manifest can be seen in Listing 5.6. In principle, three minor changes must be added to the service manifest file:

1. The Service must be deployed to the `workloads` namespace to ensure the DHRT manages only the intended Pods.
2. A `runtime` annotation must be provided, which specifies the target execution deadline for a workload.
3. The `schedulerName` property must be set to configure HDS as the scheduler instead of the default kube-scheduler.

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello-world-service
  namespace: workloads # (1)
spec:
  template:
    metadata:
      annotations:
        runtime: "100s" # (2)
    spec:
      schedulerName: deadline-scheduler # (3)
      containers:
      - image: hello-world:latest
        ports:
        - containerPort: 8080
```

Listing 5.6: **DHRT Workload Deployment.** The listing shows an example Knative Service manifest to describe a workload managed by DHRT. The required properties and developer inputs are highlighted via comments for reference.

6. Evaluation

In this chapter, the developed deadline-aware scheduling and resource allocation scheme is systematically evaluated through a series of experiments conducted on GKE using synthetic workloads. The primary goal of this evaluation is to assess the effectiveness of DHRT in meeting workload deadlines while maintaining efficient resource utilisation. These aspects are evaluated throughout different scenarios and load conditions.

While deadline adherence is a key metric throughout the experiments, it should not be interpreted in isolation. Other important factors, such as resource efficiency, SLO reliability and the magnitude of violations must also be considered for a more comprehensive evaluation. Therefore, rather than analysing these metrics in isolation, it is important to keep a holistic view to identify broader trends that reveal the strengths and limitations of different methods.

For a detailed description of the experimental setup, including the cluster configurations, workload characteristics and load generation method, refer to Section 4.3.

6.1. Overview

Experiment 1 (Section 6.2) evaluates the quality and impact of resource allocations on deadline adherence and resource efficiency. To do so, experiments are conducted on a homogeneous cluster with identical workloads but varying deadlines. The objective is to assess the effectiveness of in-place resource tuning to maintain deadlines (RQ1), as well as the accuracy of resource estimates using a heuristic-based approach (RQ2.

Experiment 2 (Section 6.3) examines the developed approach within a heterogeneous cluster using more diverse workloads but pre-determined resource allocations. The primary focus lies on evaluating the ability to utilise node heterogeneity and adapt scheduling and resource configurations of workloads to maintain SLOs, as specified in RQ3.

The overreaching goal of both experiments is the evaluation of DHRT against baseline scheduling and resource provisioning policies to answer RQ4. Specifically, the objective is to highlight commonly found shortcomings of these methods and benchmark the performance of DHRT. Namely, comparisons are performed against the default `kube-scheduler` and two resource allocation policies commonly used by FaaS providers, as well as one adaptation thereof:

- **Fixed CPU-Allocation Policy** (`P-Fixed`): Each workload is always assigned 1

vCPU, with memory being configured separately.

- **Proportional CPU-Allocation Policy** (`P-Prop`): vCPU is assigned proportionally to memory, using values provided by Google Cloud Run [54] as reference.

- **P-Prop using intelligent over-provisioning** (`P-Prop*`): Identical to `P-Prop`, but memory is over-provisioned to increase the vCPU-share until the desired performance is achieved.

6.2. Resource Allocation and Deadline Adherence within Homogeneous Clusters

The following results are based on experiments performed on a GKE cluster in region `europe-central2` using the node configuration from Table 4.1a. For this experiment, a homogeneous cluster is utilised to isolate and focus on resource allocation decisions and their impact on deadline adherence.

The objective is to evaluate how effectively DHRT can determine resource configurations that reliably maintain deadlines and how it compares to the previously described baseline policies. With limited resources available, resource allocations must reliably fulfil SLOs while avoiding significant over-provisioning. Therefore, different load scenarios are evaluated throughout the experiment. The following sections first present the results of the baseline resource allocation policies, highlighting their commonly found limitations. Then, the results of DHRT are showcased and evaluated in comparison.

To assess the quality of resource allocations, the `Power-Iteration` workload is deployed with varying deadlines, representing different performance needs for a compute-bound workload as described in Section 4.3. We define three levels of deadlines for the workload based on a baseline execution time of 135 seconds, which was determined through profiling on an N1-Node using 1 vCPU:

Workload Profile	Deadline	N	Iterations
Strict	108s (80% of baseline)	10000	1000
Moderate	155s (115% of baseline)	10000	1000
Relaxed	270s (200% of baseline)	10000	1000

Table 6.1.: **Experiment 1 - Workload Configurations.** Three identical workloads are deployed, with varying deadlines representing different performance requirements.

6.2.1. Fixed CPU-Allocation Policy

Due to its static resource allocation of 1 vCPU, `P-Fixed` is inherently inflexible and cannot be adapted to different workload or cost requirements. While this simplicity is often sufficient, it cannot reliably fulfil SLOs as it lacks adaptability to tune resource allocations.

This limitation becomes evident in the experiment results shown in Figure 6.1, where `Workload- Strict` consistently fails to meet its execution deadline as a result of under-provisioning. In contrast, `Workload-Relaxed` never violates its specified deadline, however it consistently finishes its execution well ahead of the deadline. While this would not inherently seem like a bad property, this constant over-provisioning can result in higher operational costs or resource starvation in fixed-size clusters.

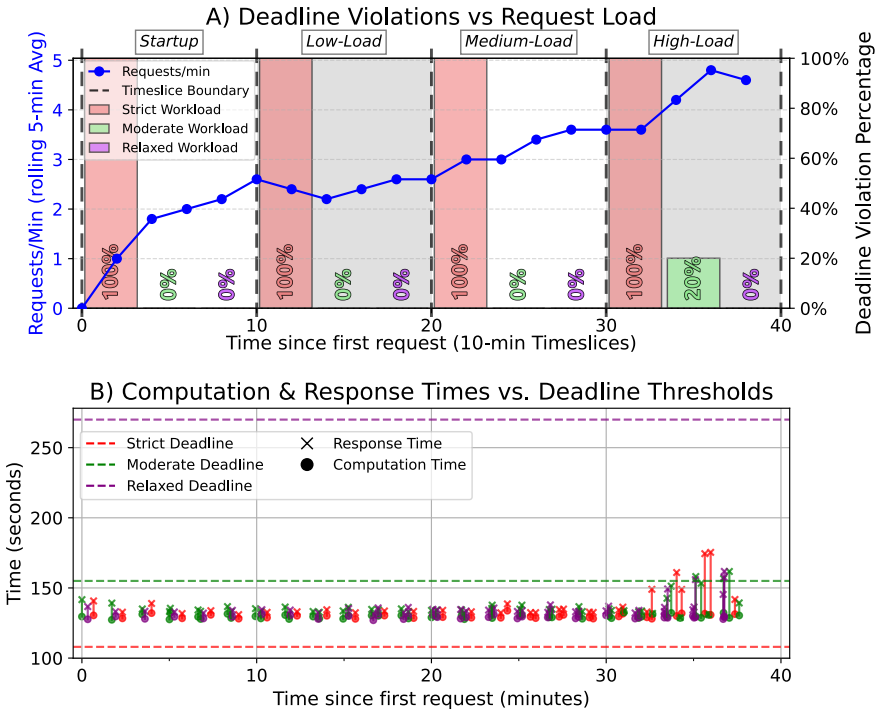

Figure 6.1.: **Results P-Fixed.** A static allocation of 1 vCPU is unable to fulfil different performance requirements, resulting in equal execution times regardless of the specified deadline. **Figure A** shows the aggregated deadline violations throughout different request load levels. **Figure B** shows a more granular view, focusing on individual workload response and computation times.

The consequence of this resource inefficiency can be observed within the 30-40 minute period (High-Load), where the increased number of workload invocations leads to significant scheduling delays due to resource starvation. This is indicated by the noticeable difference between the workload's response time and computation time. Particularly, for

the `Workload-Moderate` this results in deadline violations in 20% of invocations.

6.2.2. Proportional CPU-Allocation Policy

`P-Prop` is a widely used resource allocation scheme for FaaS offerings, where customers select a memory configuration from a predefined set of options. The corresponding vCPU allocation is then assigned proportionally. For example, our experimental workload requires approximately 870 MiB of memory, therefore the configuration `[1024 MiB, 0.583 vCPU]` is selected.

These values are based on the configuration model of Google Cloud Run [54], which assumes a fixed relationship between vCPU and memory allocation to accommodate typical workloads. However, this allocation strategy inherently fails to account for workloads with more diverse resource profiles or specific performance targets. As a result, CPU-intensive workloads are often under-provisioned, leading to insufficient performance and deadline violations. Conversely, memory-intensive workloads may receive more vCPU than required, leading to inefficient resource utilisation and increased costs for the user.

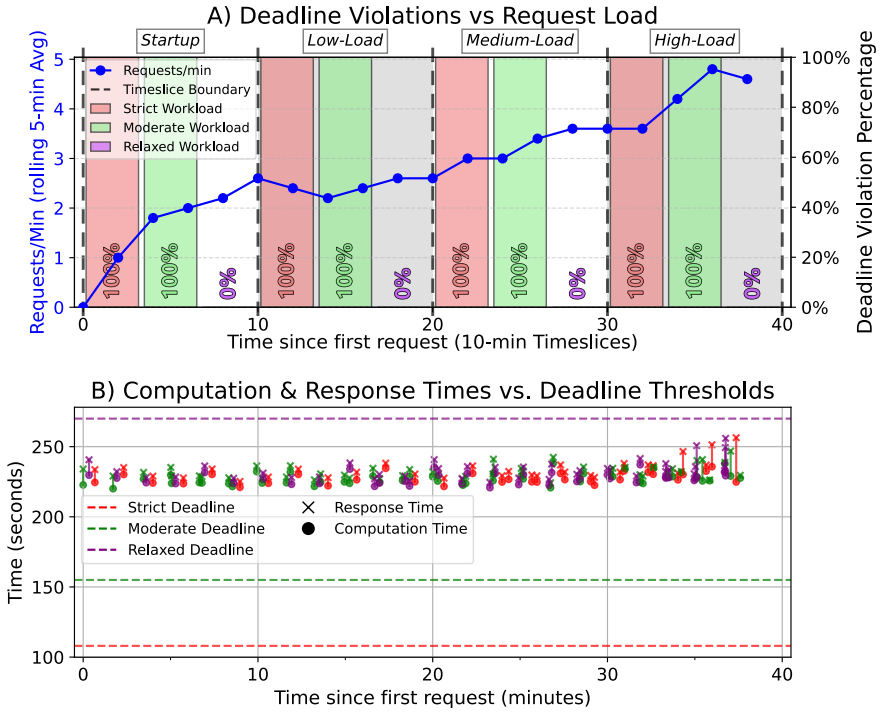

Figure 6.2.: **Results P-Prop. Figure A** highlights that a proportional allocation scheme fails to capture the performance requirements of different workloads, meeting only the deadline of `Workload-Relaxed`. **Figure B** shows that an allocation of merely 0.583 vCPU leads to execution times significantly exceeding the deadline thresholds for `Workload-Moderate` and `Workload-Strict`.

Looking at the results presented in Figure 6.2, it can be observed that both `Workload-Strict` and `Workload-Moderate` are unable to finish execution within their specified deadline. This is because `P-Prop` underestimates the vCPU requirements of these workloads, which causes them to miss their performance targets. Only `Workload-Relaxed` is able to terminate within its execution deadline as its resource requirements align more closely with the provided resource configuration.

Over-provisioning Strategy

A common strategy to achieve higher performance and circumvent the limitations of `P-Prop` is to over-provision memory, as this indirectly increases the allocated vCPU. Essentially, this over-provisioning strategy (`P-Prop*`) trades resource efficiency for improved performance. By choosing higher memory configurations for `Workload-Moderate` (`[2048 MiB, 1 vCPU]`) and `Workload-Strict` (`[4096 MiB, 2 vCPU]`), it enables all workloads to complete within their specified deadlines under low-load conditions, as shown in Figure 6.3.

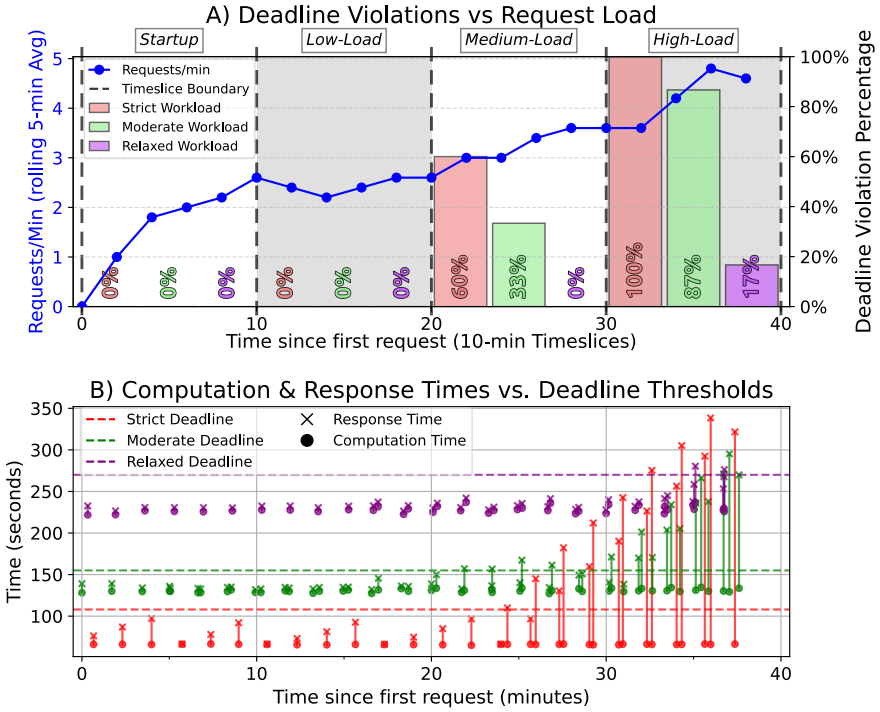

Figure 6.3.: **Results P-Prop*.** The over-provisioning strategy `P-Prop*` achieves the required performance under low request loads. However, due to excessive resource allocations, it leads to significant deadline violations as request load increases, as shown in **Figure A**. Meanwhile, **Figure B** highlights significant deviations between computation and response times.

However, this strategy comes at the cost of significant resource inefficiency, which becomes

evident in the second half of the experiment as the request load increases. As a result of the resource over-allocation, new workload invocations can no longer be accommodated, leading to significant scheduling delays and thus missed deadlines for all workloads.

Overall, proportional allocation schemes offer more flexibility than fixed allocation methods. Nevertheless, these policies inherently involve a trade-off between operational efficiency and performance for most workloads, which becomes more severe as workloads deviate further form the available vCPU-to-memory proportions. Furthermore, developers cannot specify whether they want to prioritise performance or cost-efficiency; instead, they must manually determine the configuration that most closely matches their requirements. These limitations make proportional schemes unsuitable to fulfil fine-grained SLOs, particularly for workloads with diverse resource profiles.

Therefore, it is essential to find targeted resource allocations that are both efficient, minimising resource waste, while still reliably fulfilling SLOs, even under high-load conditions. This challenge is a key motivation behind DHRT, which is discussed in the following section. Additionally, to further illustrate the difference in resource efficiency of `P-Prop*` and DHRT, a comparison of workload resource allocations is provided in Table 6.2.

6.2.3. DHRT: Deadline-Aware Resource Allocation Strategy

The deadline-based resource allocation strategy of DHRT aims to overcome the shortcomings of previous policies by dynamically tuning allocations according to workload requirements. As a result of this, it is able to achieve consistently lower deadline violations, as showcased in Figure 6.4.

Initially, a high variability in execution times can be seen for the first few workload invocations. This is because DHRT does not have any prior information about the workloads and must first learn their resource requirements. Therefore, a default allocation of `[2048 MiB, 1 vCPU]` is initially used for all workloads.

As a result of this uncertainty, it leads to some initial deadline violations for `Workload-Strict` due to under-provisioning, whereas `Workload Relaxed` is over-provisioned. However, within a few samples DHRT is able to correct this and adapt its resource estimates according to individual workload demands. This property can be observed in Figure 6.5, where HRE is able to quickly fine-tune its resource estimates to find more reliable and efficient resource allocations over time by learning from past execution data. However, it should be noted that due to workloads often executing concurrently, the impact of these improved estimates is not always immediate.

While DHRT also experiences performance degradation during peak request load, the impact on deadline violations can be kept minimal. This robustness of DHRT in high-load conditions can be attributed primarily to two factors, which are subsequently discussed.

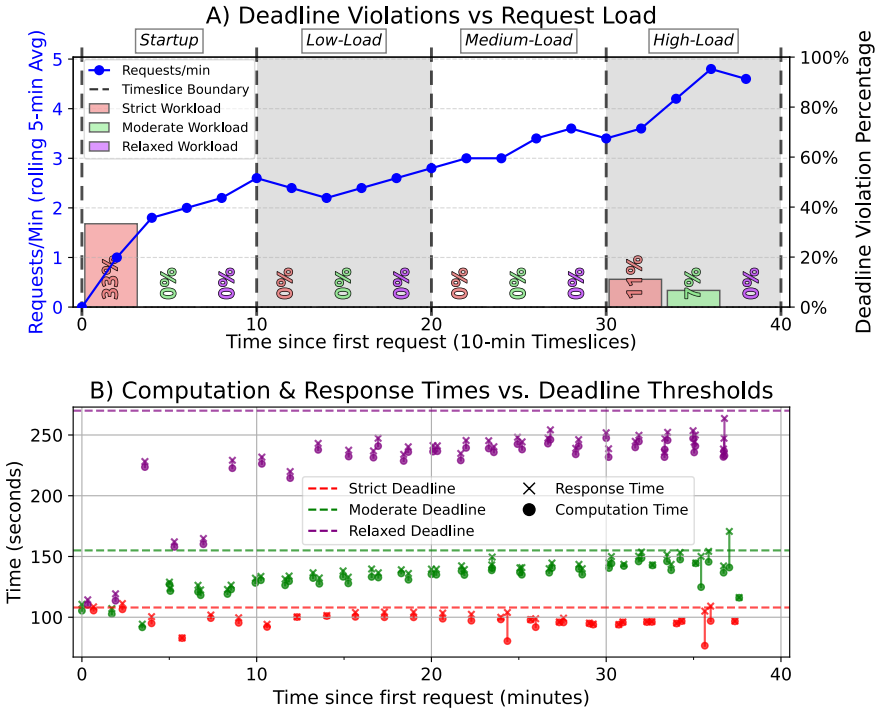

Figure 6.4.: **Results DHRT.** During the startup period, `Workload-Strict` experiences some deadline violations, which are corrected for later executions as workload requirements become known. Between 30-40 minutes, slight violations occur due to scheduling delays from high request load.

Adaptive Resource Allocation and Efficiency

First, DHRT attempts to find vCPU configurations that target runtimes around 90% of the specified deadline. This property can be observed in Figure 6.4 (B), with response times very closely approaching the respective deadline thresholds. By accurately considering workload requirements and avoiding over-provisioning, this strategy can achieve higher resource efficiency while maintaining SLOs more reliably.

The importance of resource efficiency becomes evident when comparing the resource allocations with the previously discussed `P-Prop*`, as seen in Table 6.2. While both strategies can fulfil performance requirements during low-load conditions, `P-Prop*` cannot maintain SLOs as the request load grows. In contrast, DHRT can adapt its resource allocations to the workload requirements, which results in significantly more efficient allocations by only reserving necessary resources for a workload.

In the most severe case of `Workload-Strict`, DHRT achieves 35.8% lower vCPU allocations and 79.7% lower memory allocations compared to `P-Prop*`. This over-provisioning

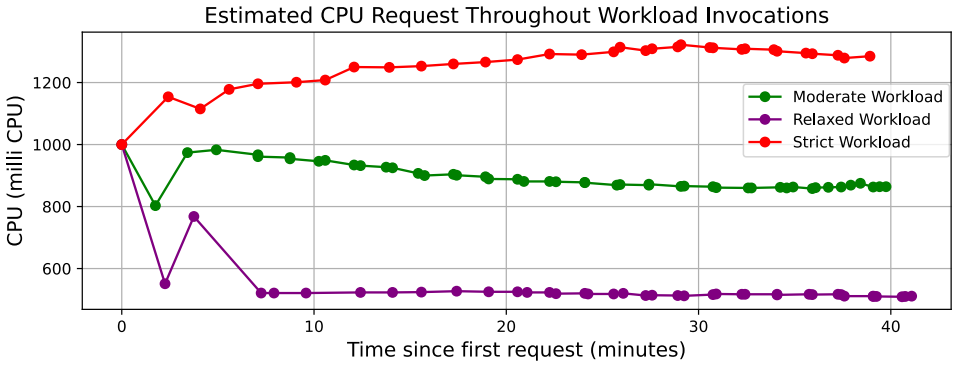

Figure 6.5.: **DHRT CPU-Request Estimates Over Time.** The figure shows how HRE refines its vCPU-allocation estimates for subsequent workload invocations as more information becomes available.

results in significant SLO degradation for `P-Prop*` during high-load periods, whereas DHRT is able to maintain deadlines in 89%, 93% and 100% of cases for the respective workloads.

Workload Profile	`P-Prop*`			`DHRT`		
	CPU Request (mCPU)	**Memory Request (MiB)**	**Deadline Fulfilled (High-Load)**	**CPU Request (mCPU)**	**Memory Request (MiB)**	**Deadline Fulfilled (High-Load)**
Strict	2000	4096	0%	1285	831	89%
Moderate	1000	2048	13%	864	830	93%
Relaxed	583	1024	83%	511	830	100%

Table 6.2.: **Resource Efficiency Comparison for `P-Prop*` and `DHRT`.** The table compares the used resource requests and deadline adherence for both strategies during high-load periods for each workload profile.

Deadline-Aware In-place Vertical Scaling

Secondly, DHRT is able to reduce the risk of deadline violations, particularly during high-load periods through reactionary measures implemented by IRT, using in-place vertical Pod scaling. By dynamically increasing the allocated vCPU-share of workloads as they closely approach their deadline, more reliable execution times can be achieved. This helps to lower the impact of performance variability common in virtualised and shared environments.

Furthermore, this dynamic vertical scaling can help to counteract significant scheduling

delays caused by resource starvation. If such delays are detected, the vCPU-allocation of the affected workload is dynamically boosted by IRT relative to the delay to accelerate computation and try to stay within the deadline threshold. This behaviour is particularly noticeable around the 35-minute mark in Figure 6.4 (B), where significantly lower computation times can be observed for several workloads to maintain response times below the deadline threshold. By applying these dynamic resource adjustments, DHRT can adapt to high-load conditions where resource contention is high and maintain SLOs more reliably. Nevertheless, this adjustment relies on sufficient resources being available on a node and can not always be applied.

6.2.4. Summary of Findings

This section provides a brief summary of key findings, comparing the baseline resource allocation policies `P-Fixed, P-Prop and P-Prop*` with DHRT, highlighting individual strengths and limitations.

Table 6.3 presents a comparison of key metrics, focusing on deadline adherence and the extent to which the average response time deviates from the target deadline across all workloads. This metric is particularly valuable as it can provide insights into potential under-provisioning or over-provisioning of resources. Ideally, deadline adherence should be maximised, while runtime deviations are kept minimal to ensure both efficiency and reliability.

Policy	**Relaxed**		**Moderate**		**Strict**	
	Deadline Fulfilled (%)	**Average Deadline Dev. (%)**	**Deadline Fulfilled (%)**	**Average Deadline Dev. (%)**	**Deadline Fulfilled (%)**	**Average Deadline Dev. (%)**
`P-Fixed`	100.00	-49.36	89.58	-7.57	0.00	28.43
`P-Prop`	100.00	-13.38	0.00	56.87	0.00	116.62
`P-Prop*`	94.87	-12.03	58.33	10.14	51.61	39.90
`DHRT`	100.00	-14.02	97.92	-11.29	90.32	-7.69

Table 6.3.: **SLO-Comparison of Resource Allocation Strategies**. The result metrics compare the overall percentage of maintained deadlines as well as the average runtime deviation from the deadline, where positive values ($> \mathbf{0}$) indicate deadline violations, across workload types (Relaxed, Moderate, Strict).

`P-Fixed`: Performs particularly well for `Workload-Moderate`, achieving a deadline adherence of 89.58%. This is because the deadline was selected based on execution times using 1 vCPU, which exactly matches the strategy of `P-Fixed`. However, this policy struggles when workloads require higher performance levels, which leads to under-provisioning and missing all deadlines for `Workload-Strict`. In contrast, `Workload-Relaxed` always

terminates within its deadline but has a large average runtime deviation of -49.36%, which indicates inefficient resource utilisation due to over-provisioning.

`P-Prop:` Effective when the allocated proportions closely match workload requirements, as seen with `Workload-Relaxed` which always terminates in time with only a minor deviation from the target runtime of -13.38%. For stricter deadlines, which demand a higher vCPU-to-memory ratio, the strategy fails to provide sufficient performance, resulting in severe deadlines violations due to under-provisioning.

`P-Prop*:` In principle, this strategy is able to maintain all workload deadlines by over-provisioning memory until the allocated vCPU matches the performance requirements. However, this comes at the cost of significant resource inefficiency due to excessive resource allocations. Consequently, this resource contention leads to a sharp increase in scheduling delays during high-load periods, resulting in significant SLO degradation, with deadline adherence dropping to 58.33% and 51.61% for `Workload Moderate, Strict` respectively.

`DHRT:` Directly incorporates workload characteristics and SLO requirements into resource allocations decisions, allowing it to consistently outperform baseline policies across all workload types. By iteratively refining allocations as more information becomes available, combined with dynamic resource adjustments, it can reliably fulfil execution deadlines while mostly avoiding under- or over-provisioning. Overall, these properties allow DHRT to maintain deadline adherence above 90% while keeping the average runtime very close to the specified targets, with deviations ranging from -14% to -8%.

6.3. Heterogeneity-Aware Scheduling and Resource Adaptation

The second experiment was conducted with the `europe-west4` region on GKE, using a heterogeneous cluster based on the configuration described in Table 4.1b. This evaluation primarily addresses RQ3, assessing how well DHRT can align diverse workload requirements with the available node capabilities for scheduling and resource allocation decisions. As a baseline for comparison, Kubernetes' default `kube-scheduler` is used.

To conduct the evaluation, three different versions of the `Power Iteration` workload were deployed, each with a distinct resource profile, as shown in Table 6.4. While the first experiment focused on determining efficient resource allocations based on deadlines, this is not the primary objective in this case. Instead, workloads are configured with pre-determined resource allocations that reliably meet the specified deadline. Specifically, these allocations are based on the average execution time on an N1 node under the given configuration, with an additional 10% added to the deadline to accommodate performance variations.

This means that under normal load conditions, arbitrary scheduling decisions should

Workload Profile	Deadline	N	Iterations	vCPU	Memory
Compute	130s	10000	1500	1700 mCPU	896 MiB
Balanced	112s	12000	400	800 mCPU	1280 MiB
Memory	148s	20000	150	600 mCPU	3584 MiB

Table 6.4.: **Workload Profiles and Resource Configurations.** The table outlines the different workload profiles used in the heterogeneous cluster experiment. Each profile has a different balance of vCPU and memory requirements. The deadline was set according to the observed average execution time on an N1 node in GKE with an additional 10% buffer.

still reliably meet deadlines on any node. However, under high-load scenarios, this experiment highlights how deadline adherence and resource efficiency can be improved by incorporating node capabilities into scheduling and resource allocation decisions.

6.3.1. Results

The experiment results for both the baseline `kube-scheduler` as well as DHRT are shown in Figure 6.6 and 6.7 respectively, with key SLO-metrics summarised in Table 6.5.

For the `kube-scheduler`, deadline violations are naturally absent during low-load periods due to the pre-configured allocations of vCPU and memory. However, as soon as the request load starts to increase, a large rise in the difference between response and computations times can be observed. This indicates that workloads were pending for longer periods due to insufficient resources. Consequently, the number of deadline violations increases to 31-38%, with the volatility of response times increasing significantly, resulting in very severe deadline violations for workloads that cannot be scheduled immediately.

For DHRT, a clear learning period can be observed during the first few executions of each workload. While the initial resource allocations were set to predetermined values, no additional information was provided. As we typically do not deal with workloads where such detailed knowledge is available beforehand, DHRT still treats these initial estimates with a high degree of uncertainty. This leads to higher variability in execution times initially as resource allocations are more frequently adjusted.

While DHRT also experiences SLO degradation during the highest load period (20-30 min), the impact is significantly less severe. In several cases, deadline violations are mitigated by dynamically tuning the allocated vCPU to accelerate computation and offset scheduling delays. This capability is especially beneficial to recover from such scenarios where the expected performance cannot be maintained due to resource contention.

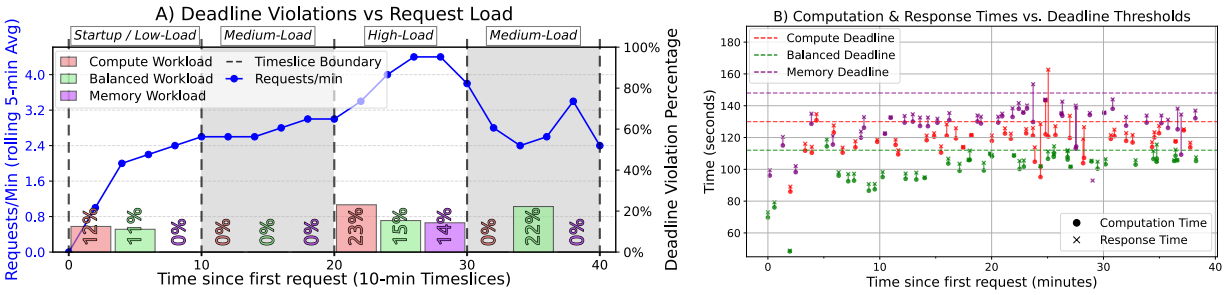

Figure 6.6.: **Results DHRT. Figure A** highlights that by incorporating the underlying heterogeneity into scheduling and resource allocation decisions, DHRT achieves improved deadline adherence. Meanwhile, **Figure B** shows that by accounting for the available node capabilities, DHRT achieves significantly more consistent execution times in addition to fewer and less severe deadline violations.

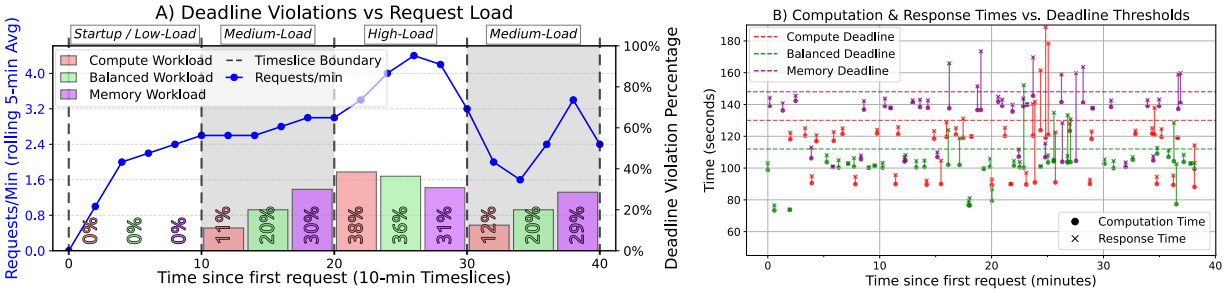

Figure 6.7.: **Results kube-scheduler. Figure A** shows that the kube-scheduler struggles to maintain deadline under high-load conditions, resulting in deadline violations in approximately one third of workload invocations. Significant variability in execution times can be seen in **Figure B**, as it does not adapt its resource allocations according to the node selected during scheduling, failing to capture different machine capabilities.

6.3.2. Discussion

Overall, the previously described results show that DHRT is able to achieve significantly more reliable performance when compared to the `kube-scheduler`. This can be attributed to three primary differences, which are subsequently discussed.

Heterogeneity Awareness

The `kubes-scheduler` fails to capture the physical heterogeneity available within the cluster. This becomes evident in Figure 6.7, where a subset of workload invocations have much faster execution times, terminating well before their specified deadline. This is

because the same resource allocation is used for all nodes, leading to over-provisioning on the more powerful C3 node.

Contrary to this, DHRT dynamically incorporates node capabilities and updates its resource allocations accordingly. Figure 6.8 shows that it immediately adapts its resource allocations for the performance-optimised C3 node as workload information is collected. This highlights that due to its heterogeneity awareness, DHRT can achieve significant resource efficiency gains. Compared to the `kube-scheduler`, which uses the same resource allocation for all nodes, DHRT is able to reduce the required vCPU allocation on the C3 node by approximately 28% across all workloads while maintaining SLOs. These efficiency improvements are particularly beneficial in high-load scenarios, since more workloads can be scheduled simultaneously while reducing resource contention.

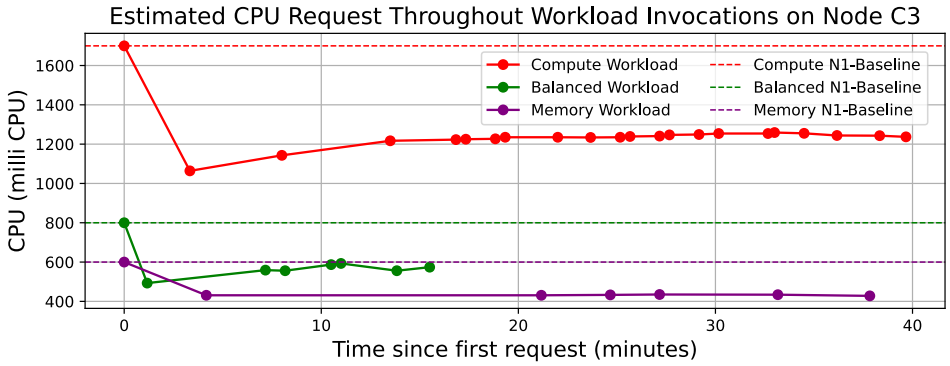

Figure 6.8.: **DHRT CPU-Request Estimates Over Time on C3.** The figure shows how HRE refines its CPU-Allocation estimates to incorporate the higher compute capability of the performance-optimised C3 node as more information becomes available. The final estimates are approximately 28% lower when compared to the baseline used on the N1-node.

Resource-Aware Scheduling

The `kube-scheduler` primarily makes scheduling decisions based on the currently available vCPU and memory, prioritising low-utilisation nodes and targeting a balanced resource usage. While this approach effectively distributes workloads in the short term, it can lead to inefficiencies in heterogeneous environments over time. Specifically, placing a CPU-intensive workload on a high-memory node or vice versa can increase the risk of resource fragmentation. Such one-sided resource usage patterns can reduce scheduling flexibility, making it more difficult to place future workloads efficiently.

Scoring in DHRT follows a similar approach to the `kube-scheduler` but also tries to consider the long-term impact of workload placement. Specifically, it actively matches workload resource requirements with suitable node types to optimise resource efficiency.

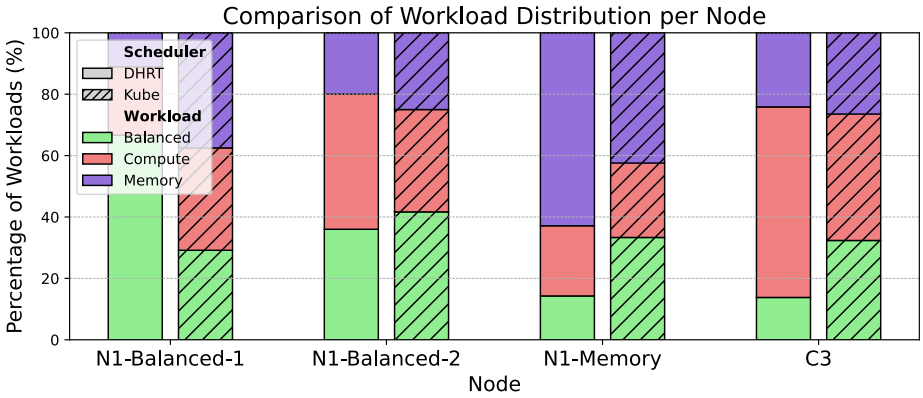

Figure 6.9.: **Distribution of Workloads to Nodes.** The figure compares the placement of workloads between DHRT and the kube-scheduler throughout the experiment, showing their allocation across the two general-purpose N1 nodes, the high-memory N1 variant and the compute-optimised C3 node.

A comparison between different workload distributions across nodes is shown in Figure 6.9. It can be observed that while the `kube-scheduler` places workloads in a highly balanced way regardless of the node, DHRT shows a bias to schedule workloads to nodes with complementary resources. By introducing the Resource-Alignment Score, a more targeted placement strategy can be achieved which favours placing memory-intensive workloads on the high-memory N1 node and compute-intensive workloads on the C3 node.

Deadline Awareness

Lastly, by closely monitoring workload executions and performing dynamic vCPU scaling as workload approach their deadline, DHRT is much more robust during challenging conditions. Unlike static resource allocation strategies, its adaptive approach based on real-time feedback can significantly reduce deadline violations. As previously observed, this is particularly valuable during periods of high resource load, where performance variations and scheduling delays are more likely to occur.

6.3.3. Summary of Findings

The evaluation results highlight the advantages of DHRT in maintaining execution deadlines under varying workload conditions. Looking at the direct comparison in Table 6.5, the results show that while both methods maintained similar average response times within a 5% margin, the `kube-scheduler` had a significantly higher variability as indicated by the increased standard deviation. This effect is especially noticeable during high-request periods, as outlined in Table 6.5b.

A major reason for this is the `kube-scheduler`'s inability to effectively utilise the performance-optimised C3 node, leading to reduced resource efficiency. Moreover, DHRT's ability to offset delays through dynamic adjustments can help to mitigate the severity of SLO violations.

Lastly, it should be noted that the utilised cluster had a very moderate degree of both physical and virtual heterogeneity. A larger variety of machine families and virtual resource configurations could further potential to improve resource efficiency and deadline adherence through heterogeneity-aware resource tuning.

Method	**Compute**		**Balanced**		**Memory**	
	Deadline Fulfilled (%)	**Response Time** (μ, σ)	**Deadline Fulfilled** (%)	**Response Time** (μ, σ)	**Deadline Fulfilled** (%)	**Response Time** (μ, σ)
`kube-scheduler`	81.58	120.80 s, 22.14 s	80.00	106.56 s, 15.14 s	75.68	138.59 s, 19.56 s
`DHRT`	90.00	122.62 s, 10.39 s	87.80	102.50 s, 12.30 s	97.50	132.14 s, 11.45 s

(a) Results across the whole experiment.

Method	**Compute**		**Balanced**		**Memory**	
	Deadline Fulfilled (%)	**Response Time** (μ, σ)	**Deadline Fulfilled** (%)	**Response Time** (μ, σ)	**Deadline Fulfilled** (%)	**Response Time** (μ, σ)
`kube-scheduler`	61.54	129.38 s, 31.13 s	63.64	115.23 s, 18.58 s	69.23	142.87 s, 16.81 s
`DHRT`	76.92	128.90 s, 11.88 s	84.62	107.84 s, 3.82 s	85.71	136.22 s, 12.98 s

(b) Results during high-request load (t=20-30 min).

Table 6.5.: **SLO-Comparison of kube-scheduler and DHRT**. The table compares the results of both methods with regards to deadline adherence and the average response time and standard deviation of individual workloads. A high standard deviation indicates higher workload performance volatility, typically caused by increased scheduling delays.

7. Conclusion

This chapter concludes the thesis by summarising key contributions and findings related to the proposed research questions. It also discusses encountered challenges as well as learnings throughout the thesis and provides an outlook for possible future work.

7.1. Contributions and Findings

The goal of this thesis was to develop a heterogeneity-aware scheduling and resource allocation solution which directly incorporates user-specified deadlines as primary SLO. With workloads and cloud resources becoming increasingly diverse, the proposed approach attempts to leverage this heterogeneity to enable developers to deploy performance-critical serverless workloads without the need for exhaustive manual tuning of resource allocations.

Evaluations showed that the developed system, DHRT, can effectively estimate efficient resource configurations that can reliably maintain deadlines within a small number of live workload executions using a heuristic-based approach. Additionally, by dynamically adjusting resource allocations and leveraging real-time execution feedback, DHRT achieves improved deadline adherence while avoiding over-provisioning. This adaptive approach significantly reduces the risk of SLO violations during periods of high resource contention, where performance degradation and scheduling delays are more likely to occur. Lastly, in-place vertical scaling can help to correct bad initial resource estimates common in online optimisation settings.

Furthermore, a key advantage of DHRT is its ability to exploit heterogeneous node capabilities to make more informed scheduling and resource allocation decisions. Unlike traditional schedulers, which use a static resource allocation for workloads across all nodes, DHRT adapts its decisions based on the performance characteristics of different node types. The evaluation demonstrated that this approach can significantly improve vCPU allocations, reducing resource waste and resulting in higher resource efficiency. Naturally, this becomes increasingly important as performance differences between available nodes grow larger.

Overall, these initial evaluations highlighted the potential of such dynamic resource tuning methods, especially in virtualised resource environments where performance variations are common. By utilising real-time metrics and adapting resource allocation and scheduling decisions according to the resource requirements of workloads, both deadline adherence and cluster-wide resource efficiency can be improved. By doing so, this approach

can shift the serverless deployment model from a best-effort strategy to an SLO-driven model, prioritising workload-specific performance guarantees.

7.2. Challenges and Learnings

Developing DHRT presented several challenges, specifically as it relies on novel and still experimental Kubernetes features, such as `InPlacePodVerticalScaling`. While it offers many promising opportunities, its early development status introduced several challenges, including compatibility issues and limited documentation. As a result, extensive testing was required to obtain reliable results.

Another challenge was to reliably isolate individual request metrics while keeping user-applications a black-box. Since Kubernetes' abstraction layer primarily deals with Pods, it can be difficult to track distinct workload executions. This lack of request awareness also led to another problem where arbitrary Pods were deleted during scale-down operations, thus occasionally terminating Pods with an active workload. To address these challenges, Knative's sidecar Queue-Proxy was adapted to provide accurate metrics on a request-level and extended to use the `PodDeletionCost` beta feature to ensure deterministic Pod deletions.

Additionally, it proved to be quite challenging find a good balance between dynamic vCPU scaling and determining good initial allocations based on past execution data. Since these factors indirectly influence each other, underestimated resource estimates were sometimes not penalised enough since they could often be corrected by in-place vCPU scaling. However, this could lead to deadline violations if not enough resources were available on the node to facilitate vertical scaling. Therefore, to improve performance stability, dynamic adjustments should gradually become less frequent as resource estimates become more accurate. Once enough execution data is available, in-place scaling should only be applied when absolutely needed to maintain performance.

7.3. Future Work

This thesis explored the opportunities of deadline- and heterogeneity-aware resource tuning in a serverless setting. However, it primarily focuses on compute-bound workloads with consistent behaviour across different invocations. A natural next step would be to explore more diverse workloads, such as input-dependent workloads, which require more fine-grained resource estimates and dynamic adjustments. Additionally, due to its growing relevance and shared workload characteristics, AI-inference could be a key candidate to investigate the applicability of DHRT for real-world workloads.

Incorporating cluster scaling strategies could further optimise costs and help mitigate SLO violations under high request loads. This could involve not only adjusting the number of nodes but also dynamically selecting the right node types to meet performance

targets in a cost-efficient way. Such an approach could even allow developers to define a preferred trade-off between performance and cost.

Finally, exploring different resource optimisation and automation techniques could bring further benefits. The current approach relies primarily on heuristics and real-time scaling, however, future work could investigate the effectiveness of machine-learning based models for workload classification and resource predictions. This could enable more proactive scaling, especially for workloads with more complex usage patterns, such as distinct computational and I/O phases. By recognising these phase transitions, the system could dynamically adjust resources to ensure efficient execution without over-provisioning.

List of Tables

2.1. Technology Stack 5

3.1. Summary of Reviewed Related Work 28

4.1. Cluster configurations used during the evaluation 33
4.2. Experimental Workload Metrics 35

5.1. Frequency of In-Place Pod Resizing 48

6.1. Experiment 1 - Workload Configurations. 64
6.2. Resource Efficiency Comparison for `P-Prop*` and `DHRT` 70
6.3. SLO-Comparison of Resource Allocation Strategies 71
6.4. Experiment 2 - Workload Configurations. 73
6.5. SLO-Comparison of kube-scheduler and DHRT 77

A.1. Example Execution Profiles 102

List of Figures

2.1. Kubernetes Cluster Architecture 8
2.2. Kubernetes Scheduling Framework Overview 11
2.3. Cloud Computing Abstraction Layers 15
2.4. Knative Serving Components 16

4.1. System Model Overview 30

5.1. High-Level System Overview 38
5.2. Sequence-Diagram: System Workflow and Component Interactions 39
5.3. IRT Upscaling Magnitude. 49

6.1. Experiment 1 - Results Fixed CPU-Allocation (P-Fixed) 65
6.2. Experiment 1 - Results Proportional CPU-Allocation (P-Prop) 66
6.3. Experiment 1 - Results Proportional CPU-Allocation using Over-provisioning (P-Prop*) 67
6.4. Experiment 1 - Results DHRT. 69
6.5. Experiment 1 - DHRT CPU-Request Estimates Over Time. 70
6.6. Experiment 2 - Results DHRT 74
6.7. Experiment 2 - Results kube-scheduler 74
6.8. Experiment 2 - DHRT CPU-Request Estimates Over Time. 75
6.9. Experiment 2 - Distribution of Workloads to Nodes. 76

Listings

2.1. Pod Manifest with Resource Requests and Limits 9
2.2. Example Knative Serving Deployment Manifest 16

5.1. Pod Resizing Logic 48
5.2. Example ConfigMap of a Pod's Resource Estimates. 56
5.3. Configuration of cAdvisor ServiceMonitor 58
5.4. Example Request Log Data. 59
5.5. Grafana Alert Configuration for Queue Proxy Logs. 59
5.6. DHRT Workload Deployment. 61

A.1. KubeSchedulerConfiguration of HDS 97
A.2. Updating Knative's config-observability 98
A.3. In-place Pod Resizing via kubectl 99
A.4. In-place Pod Resizing using Kubernetes client library 99
A.5. Sample Execution Profile 100
A.6. Locust Example Response Data 103

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," Jan 2009.

[2] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," 2019.

[3] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, "With great freedom comes great opportunity: Rethinking resource allocation for serverless functions," *Proceedings of the Eighteenth European Conference on Computer Systems*, May 2023.

[4] S. Nastic, A. Morichetta, T. Pusztai, S. Dustdar, X. Ding, D. Vij, and Y. Xiong, "Sloc: Service level objectives for next generation cloud computing," *IEEE Internet Computing*, vol. 24, no. 3, p. 39–50, May 2020.

[5] I. Rocha, C. Gottel, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, "Heats: Heterogeneity-and energy-aware task-based scheduling," *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Feb 2019.

[6] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, p. 77–88, Mar 2013.

[7] Kubernetes, "Workloads," https://kubernetes.io/docs/concepts/workloads, 2023, [Online; Accessed: 18.03.2025].

[8] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2988336.2988337

[9] Docker, https://www.docker.com, 2024, [Online; Accessed: 18.06.2024].

[10] Kubernetes, https://kubernetes.io, 2024, [Online; Accessed: 20.06.2024].

[11] ——, "Cluster Architecture," https://kubernetes.io/docs/concepts/architecture, 2024, [Online; Accessed: 27.11.2024].

[12] ——, "Resource Management for Pods and Containers," https://kubernetes.io/docs/concepts/configuration/manage-resources-containers, 2024, [Online; Accessed: 27.11.2024].

[13] ——, "Autoscaling Workloads," https://kubernetes.io/docs/concepts/workloads/autoscaling, 2024, [Online; Accessed: 13.03.2025].

[14] ——, "Kubernetes Scheduler." https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler, 2024, [Online; Accessed: 03.01.2025].

[15] ——, "Kubernetes Scheduler Configuration." https://kubernetes.io/docs/reference/scheduling/config, 2024, [Online; Accessed: 04.01.2025].

[16] ——, "Scheduling Framework," https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework, 2024, [Online; Accessed: 19.07.2024].

[17] ——, "Scheduler Plugins Repository," https://github.com/kubernetes-sigs/scheduler-plugins, 2024, [Online; Accessed: 29.08.2024].

[18] ——, "Feature Gates," https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates, 2023, [Online; Accessed: 21.08.2024].

[19] ——, "Kubernetes 1.27: In-place Resource Resize for Kubernetes Pods (alpha)," https://kubernetes.io/blog/2023/05/12/in-place-pod-resize-alpha, 2023, [Online; Accessed: 21.08.2024].

[20] Knative, "Knative," https://knative.dev, 2024, [Online; Accessed: 20.06.2024].

[21] OpenFaaS, "Serverless Functions, Made Simple," https://www.openfaas.com, 2024, [Online; Accessed: 03.09.2024].

[22] Knative, "Knative Serving," https://knative.dev/docs/serving, 2024, [Online; Accessed: 22.06.2024].

[23] ——, "Knative Eventing," https://knative.dev/docs/eventing, 2024, [Online; Accessed: 22.06.2024].

[24] ——, "Knative Functions," https://knative.dev/docs/functions, 2024, [Online; Accessed: 22.06.2024].

[25] J. Chester, *Knative in Action*. O'Reilly Media, 2021.

[26] Google, "Machine families resource and comparison guide," https://cloud.google.com/compute/docs/machine-resource, 2024, [Online; Accessed: 03.10.2024].

[27] N. Mahmoudi, C. Lin, H. Khazaei, and M. Litoiu, "Optimizing serverless computing: introducing an adaptive function placement algorithm," *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, p. 203–213, 2019.

[28] A. Mampage, S. Karunasekera, and R. Buyya, "Deadline-aware dynamic resource management in serverless computing environments," *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2021.

[29] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "Ensure: Efficient scheduling and autonomous resource management in serverless environments," *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, Aug 2020.

[30] H. D. Nguyen, C. Zhang, Z. Xiao, and A. A. Chien, "Real-time serverless: Enabling application performance guarantees," *Proceedings of the 5th International Workshop on Serverless Computing*, Dec 2019.

[31] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "Cose: Configuring serverless functions using statistical learning," *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, Jul 2020.

[32] G. Safaryan, A. Jindal, M. Chadha, and M. Gerndt, "Slam: Slo-aware memory optimization for serverless applications," *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, Jul 2022.

[33] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, May 2022.

[34] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu, "Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster," *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, Dec 2017.

[35] B. Viswanathan, A. Verma, and S. Dutta, "Cloudmap: Workload-aware placement in private heterogeneous clouds," *2012 IEEE Network Operations and Management Symposium*, Apr 2012.

[36] A. Jivrajani, D. Raghu, A. KH, H. L. Phalachandra, and D. Sitaram, "Workload characterization and green scheduling on heterogeneous clusters," *2016 22nd Annual International Conference on Advanced Computing and Communication (ADCOM)*, Sep 2016.

[37] L. Thai, B. Varghese, and A. Barker, "Algorithms for optimising heterogeneous cloud virtual machine clusters," *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2016.

[38] S. Nastic, T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, D. Vii, and Y. Xiong, "Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters," *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, Sep 2021.

[39] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," *Proceedings of the 21st International Middleware Conference*, Dec 2020.

[40] A. Fuerst and P. Sharma, "Faascache: Keeping serverless computing alive with greedy-dual caching," *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr 2021.

[41] P. Vahidinia, B. Farahani, and F. S. Aliee, "Mitigating cold start problem in serverless computing: A reinforcement learning approach," *IEEE Internet of Things Journal*, vol. 10, no. 5, p. 3917–3927, Mar 2023.

[42] M. Chadha, A. Jindal, and M. Gerndt, "Architecture-specific performance optimization of compute-intensive faas functions," *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, Sep 2021.

[43] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, "Costless: Optimizing cost of serverless computing through function fusion and placement," *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, Oct 2018.

[44] T. Schirmer, J. Scheuner, T. Pfandzelter, and D. Bermbach, "Fusionize: Improving serverless application performance through feedback-driven function fusion," *2022 IEEE International Conference on Cloud Engineering (IC2E)*, Sep 2022.

[45] D. Kelly, F. Glavin, and E. Barrett, "Serverless computing: Behind the scenes of major platforms," *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, Oct 2020.

[46] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, p. 133–145, 2018.

[47] Locust, "An open source load testing tool." https://locust.io, 2024, [Online; Accessed: 02.11.2024].

[48] G. Labs, "Grafana Loki," https://grafana.com/oss/loki, 2024, [Online; Accessed: 22.08.2024].

[49] Grafana, "Grafana: The open observability platform," https://grafana.com, 2024, [Online; Accessed: 22.08.2024].

[50] Google, "Choose a minimum CPU platform," https://cloud.google.com/kubernetes-engine/docs/how-to/min-cpu-platform, 2024, [Online; Accessed: 03.12.2024].

[51] Prometheus, "Prometheus - Monitoring System & Time Series Database," https://prometheus.io, 2024, [Online; Accessed: 22.08.2024].

[52] Google, "cAdvisor - Container Advisor." https://github.com/google/cadvisor, 2024, [Online; Accessed: 19.12.2024].

[53] Kubernetes, "ReplicaSet - Pod Deletion Cost." https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#pod-deletion-cost, 2024, [Online; Accessed: 22.11.2024].

[54] Google, "Google Cloud Run Documentation." https://cloud.google.com/run/docs/configuring/services/memory-limits, 2024, [Online; Accessed: 13.12.2024].

Acronyms

API Application Programming Interface. 11, 13, 15, 47, 51

DHRT Deadline- and Heterogeneity-aware Resource Tuner. 3, 10, 29, 30, 33, 37, 38, 47, 51, 56, 57, 60, 61, 63, 64, 68–77, 79, 80, 97

FaaS Function-as-a-Service. 1, 2, 14, 17, 24, 36, 63, 66

GKE Google Kubernetes Engine. 33–35, 56, 63, 64, 72, 73, 97

HDS Heterogeneous Deadline Scheduler. 37–41, 46, 51, 61, 97

HPA Horizontal Pod Autoscaler. 9, 10

HRE Heuristic Resource Estimator. 37, 39, 40, 47, 50–52, 55, 57, 59, 60, 68, 70, 75, 97, 100

HTTP Hypertext Transfer Protocol. 17, 18

IRT Inplace Resource Tuner. 37–39, 46–48, 50, 51, 54, 57, 70, 71, 97, 100

KPA Knative Pod Autoscaler. 17, 60

OOM Out of Memory. 53, 55

SLO Service Level Objective. 1–3, 10, 21–25, 27, 29–31, 34–36, 45, 50, 54, 61, 63–65, 68–73, 75, 77, 79, 80

VM Virtual Machine. 1, 7, 22, 23, 27

VPA Vertical Pod Autoscaler. 10

A. Appendix

This chapter contains supplementary materials that provide further details about the developed software solution.

A.1. Software Project

The developed system DHRT can be found on the GitLab repository: `https://git01lab.cs.univie.ac.at/matthiasf00/dhrt`. This includes the implementation of HDS, HRE, IRT, as well as deployment manifests of all system components. Moreover, the repository includes scripts to deploy and test the system on GKE or locally using minikube. Additional information and instructions can be found in the `readme.md`.

A.1.1. Kubernetes Scheduler Configuration

The following `KubeSchedulerConfiguration` defines the scheduling profile for the developed scheduler. It registers the PreFilter, Filter, Score, and PreBind plugins to be triggered during the respective scheduling phases. To avoid unintended side-effects, some default plugins were disabled.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: deadline-scheduler-config
  namespace: resource-management
data:
  deadline-scheduler-config.yaml: |
    apiVersion: kubescheduler.config.k8s.io/v1
    kind: KubeSchedulerConfiguration
    leaderElection:
      leaderElect: false
    profiles:
    - schedulerName: deadline-scheduler
      plugins:
        preFilter:
          enabled:
            - name: deadline-scheduler
        filter:
          enabled:
            - name: deadline-scheduler
          disabled:
            - name: "NodeResourcesFit"
```

```
        score:
          enabled:
            - name: deadline-scheduler
          disabled:
            - name: "*"
        preBind:
          enabled:
            - name: deadline-scheduler
```

Listing A.1: **KubeSchedulerConfiguration of HDS.** The defined scheduling profile registers the implemented plugins within the scheduling process.

A.1.2. Queue-Proxy Request Logging

To improve observability in Knative's Queue-Proxy, the following configuration is used to enable detailed request logging. The log template includes service and Pod metadata, along with request details such as protocol, status, and latency. This configuration is required to get isolated request metrics, since Queue-Proxy by default only exposes aggregated metrics.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-observability
  namespace: knative-serving
  labels:
    serving.knative.dev/release: devel
data:
  logging.enable-var-log-collection: "true"
  logging.request-log-template: |
   {"requestMethod": "{{.Request.Method}}",
    "requestUrl": "{{js .Request.RequestURI}}",
    "status": {{.Response.Code}},
    "latency": "{{.Response.Latency}}s",
    "protocol": "{{.Request.Proto}}",
    "revisionName": "{{.Revision.Name}}",
    "namespace": "{{.Revision.Namespace}}",
    "service": "{{.Revision.Service}}",
    "configuration": "{{.Revision.Configuration}}",
    "podName": "{{.Revision.PodName}}",
    "podIP": "{{.Revision.PodIP}}"}
  metrics.backend-destination: prometheus
  metrics.request-metrics-backend-destination: prometheus
```

Listing A.2: **Updating Knative's config-observability.** The listing showcases how Knative's Queue-Proxy sidecar can be configured to log detailed request data.

A.1.3. InPlacePodVerticalScaling API

The following command demonstrates how to update the resource configuration of a running Pod using the `InPlacePodVerticalScaling` feature via kubectl. This allows to adjust resource Requests and Limits without restarting the Pod.

```
kubectl patch pod hello-world-pod -n workloads --patch '{
  "spec": {
    "containers": [
      {
        "name": "user-container",
        "resources": {
          "requests": {
            "cpu": "900m"
          },
          "limits": {
            "cpu": "950m"
          }
        }
      }
    ]
  }
}'
```

Listing A.3: **In-place Pod Resizing via kubectl.** The listing demonstrates how the CPU requests and limits of an active Pod can be updated using the `In-Place Resource Resize` feature.

While kubectl provides a manual way to update a Pod's resources, the following Go function in Listing A.4 shows how to perform the same operation using the Kubernetes API. It should be noted that Queue-Proxy side-car is explicitly included in the patch, as otherwise the resource update is not reliably applied.

```
func (pm *PodManager) patchPodResources(namespace, podName, containerName
    string, config *metrics.ResourceConfig) error {
 // Construct the patch payload using Kubernetes API types
 patch := &v1.Pod{
  Spec: v1.PodSpec{
   Containers: []v1.Container{
    {
     Name: containerName,
     Resources: v1.ResourceRequirements{
      Requests: v1.ResourceList{
       v1.ResourceCPU:    config.Requests.CPU,
       v1.ResourceMemory: config.Requests.Memory,
      },
      Limits: v1.ResourceList{
       v1.ResourceCPU:    config.Limits.CPU,
       v1.ResourceMemory: config.Limits.Memory,
      },
     },
    },
```

```
    {
     Name: "queue-proxy", // Include in PATCH due to API issues
     Resources: v1.ResourceRequirements{
      Requests: v1.ResourceList{
       v1.ResourceCPU: *resource.NewMilliQuantity(25, resource.DecimalSI)
   , // 25m CPU (default)
       v1.ResourceMemory: *resource.NewQuantity(16*1024*1024, resource.
   BinarySI), // 16 Mi
      },
      Limits: v1.ResourceList{
       v1.ResourceCPU: *resource.NewMilliQuantity(50, resource.DecimalSI)
   , // 50 mCPU
       v1.ResourceMemory: *resource.NewQuantity(128*1024*1024, resource.
   BinarySI), // 128 Mi
      },
     },
    },
   },
  },
 }

 // Marshal the patch into JSON
 patchBytes, err := json.Marshal(patch)
 if err != nil {
  return fmt.Errorf("failed to marshal pod patch: %w", err)
 }

 // Perform the patch operation
 _, err = pm.kubeClient.CoreV1().Pods(namespace).Patch(
  context.TODO(),
  podName,
  types.StrategicMergePatchType,
  patchBytes,
  metav1.PatchOptions{},
 )
 return err
}
```

Listing A.4: **In-place Pod Resizing using Kubernetes client library** The function shows how the resource Requests and Limits can be dynamically patched, as used in IRT.

A.1.4. Execution Profile Example

Below is an example of a full execution profile generated by HRE, which stores detailed resource usage and performance metrics over the course of a workload execution. This data is subsequently used to compute resource allocation estimates for future workload invocations.

```
{
  "_id": "67b5be2ff5f8812bb1caf8e9",
  "service_name": "power-iteration-moderate",
```

```
  "pod_name": "power-iteration-moderate-00003-deployment-5
   fcf5b6f76-gfhkf",
  "node_name": "gke-cluster-1-default-pool-d905c2b8-70v3",
  "node_pool": "n1",
  "status": "200",
  "time_start": "2025-02-03T11:17:13.000Z"
  "time_end": "2025-02-03T11:19:00.000Z"
  "latency": 107.76330797,
  "deadline_ratio": 0.7281304592567568,
  "cpu_metrics": {
    "usage_milli_cpu": {
      "min": 623.76,
      "max": 1529.25,
      "avg": 1081.69
    },
    "limits_milli_cpu": {
      "initial": 1050,
      "min": 1050,
      "max": 1317
    }
  },
  "memory_metrics": {
    "usage_byte": {
      "percentile_50": 869171200,
      "percentile_80": 869219532.8,
      "percentile_95": 869236736,
      "maximum": 869236736
    },
    "limits_byte": {
      "initial": 2147483648
      "min": 1260279211,
      "max": 2147483648
      }
    }
  }
}
```

Listing A.5: **Sample Execution Profile** The Listing shows an exemplary execution profile, including key metrics and metadata about a previous workload execution.

A.1.5. HRE CPU Estimate Example

The following example illustrates the computation of a CPU estimate, as described in Section 5.4.3, using the data in A.1, for a workload with a runtime deadline of 108 seconds and a soft-deadline target of 97.2 seconds (90%).

Profile	Avg CPU Usage (mCPU)	Observed Runtime (s)	Deadline Ratio	Deadline Classification
1	635.41	189.216	1.750	Hard-Violation
2	1040.21	109.512	1.014	Hard-Violation
3	1093.99	94.176	0.872	None
4	1122.56	97.416	0.902	Soft-Violation

Table A.1.: **Example Execution Profiles.** The table includes 4 execution profiles and their key metrics to derive the next CPU estimate of the workload.

Using $k = 0.2$ in (5.19) and applying the penalties in (5.20), the computed weight for each profile is approximately:

$$w_1 \approx 2.035 \times 10^{-9}, \quad w_2 \approx 0.017, \quad w_3 \approx 0.546, \quad w_4 \approx 0.766$$

The CPU request is determined by computing a weighted average of the observed average CPU usages, with the weights normalised as shown in (5.21).

$$\text{cpu}_{\text{request}} = \frac{w_1 \cdot 635.41 + w_2 \cdot 1040.21 + w_3 \cdot 1093.99 + w_4 \cdot 1122.56}{w_1 + w_2 + w_3 + w_4} \approx 1109.77 \text{ mCPU}.$$

Next, the `weighted-deadline-ratio` is computed to approximate the quality of the estimated CPU-request as shown in (5.23), using the previously computed weights and stored deadline-ratios:

$$\text{weighted-deadline-ratio} \approx 0.891$$

Subsequently, we compute the deviation from the soft-deadline target using (5.24):

$$\Delta\text{deadline-ratio} = 0.90 - 0.891 = 0.009$$

Since Δdeadline-ratio is positive, we use the first case of (5.25) to compute the steering factor:

$$\text{steering-factor} = 1 - 0.50 \cdot \Delta\text{deadline-ratio} = 1 - 0.50 \cdot 0.009 = 0.9955$$

Finally, the adjusted CPU request is computed by scaling it with the steering factor (5.26):

$$\text{cpu}'_{\text{request}} = \text{cpu}_{\text{request}} \cdot \text{steering-factor} \approx 1109.77 \times 0.995 \approx 1104.22 \text{ mCPU}.$$

This example demonstrates that profiles with runtimes closer to the soft-deadline target (in this case, Profile 3 and 4) contribute significantly more towards the final estimate than the initial two. Additionally, we can see that the steering factor has a minimal impact in this example, since the resource estimates are already very good and close to the deadline target. Therefore, no significant updates to the estimate are necessary.

A.1.6. Workload Requests and Responses

During the evaluation, important metrics are recorded via the load-testing client Locust. An example can be viewed in Listing A.6 , showing the workload's response as well as general request metrics. Additionally, the actual computation time of the workload is recorded, which makes it easy to detect delays, such as scheduling delays, by comparing it with the overall response time.

```
{
    "url": "http://power-iteration.workloads.34.91.111.166",
    "response_time": 132.621463,
    "computation_time": 128.59667205810547,
    "status_code": 200,
    "pod_name": "power-iteration-memory-6d86f9-m22g2",
    "node_name": "gke-cluster-1-n1-mem-a0dea21a-rzvh",
    "request_timestamp": "2025-01-08 19:47:11.713784"
},
{
    "url": "http://power-iteration.workloads.34.91.111.166",
    "response_time": 142.772967,
    "computation_time": 118.49354553222656,
    "status_code": 200,
    "pod_name": "power-iteration-compute-5bb8bd-dkf2s",
    "node_name": "gke-cluster-1-c3-1d612048-ztf5",
    "request_timestamp": "2025-01-08 19:47:36.716534"
},
```

Listing A.6: **Locust Example Response Data.** Two example responses are presented, which include key request metadata of the workload invocation. Based on the difference between response and computation time, a significant scheduling delay can be inferred for the second workload.