Vrije Universiteit Amsterdam

Bachelor Thesis

Task-in-Pod Scheduling Support for Kubernetes and Apache Spark Stack

Author: Karina Sudnicina (2744404)

1st supervisor:	Prof. dr. ir. Alexandru Iosup
daily supervisor:	Sacheendra Talluri, M.Sc.
2nd reader:	Dr. Daniele Bonetta

A thesis submitted in fulfilment of the requirements for the VU Bachelor of Science degree in Computer Science

November 5, 2024

Abstract

Data-intensive applications are used across different sectors, such as finance, healthcare, government, science, and entertainment. With increased data growth, special tools are needed to analyze and manage this Big Data. Examples of such tools include Apache Spark, Dask, Knative, and other frameworks. These frameworks are designed to handle large datasets by distributing processing tasks across multiple executors, either on local machines or using external tools like Kubernetes.

This project focuses on the Apache Spark framework, but the findings and methodologies are also relevant to other runtime application frameworks. When we submit an Apache Spark application within the Kubernetes cluster, a central coordinator called Spark Driver is created as a Pod within Kubernetes. Inside the Pod, Spark Driver breaks down the application into smaller tasks that can run concurrently on worker nodes. However, Kubernetes is not aware of any processes running inside the Pods, neglecting Spark Driver's task scheduling process. The problem with this lack of awareness is that it causes inefficiencies, as Kubernetes cannot adapt to the Spark Driver's task scheduling process.

To address this issue, we propose designing and developing an external scheduler, implemented as a Kubernetes Operator, to receive tasks from Spark Driver and manage their execution on worker nodes.

Our approach involves three main objectives: (i) selecting an appropriate framework for creating the Operator, (ii) designing and implementing a Task-aware scheduler, and (iii) evaluating the impact of this new scheduler on the performance of the application runtime.

In this project, we compare different Operator creator frameworks, design and implement a prototype of an external Task-aware scheduler, and evaluate its performance within the Spark-Kubernetes system using the TPC-DS benchmark workload.

Contents

1

Introduction

Every day, around 400 million terabytes of data are created, and the amount of data generated each year continues to grow exponentially [1]. This data is highly significant and used in many sectors, including healthcare, education, finance, and entertainment [2]. Storing and analyzing all this data requires substantial computational power that local computers cannot offer.

As a result, cloud computing - the fifth generation of computing technology - has become extremely popular and highly utilized in recent years [3]. Cloud computing enables widespread, convenient, on-demand network access to a shared pool of computing resources, including networks, servers, storage, applications, and services. These resources can be quickly provisioned and released with minimal management effort or interaction with service providers [4].

Despite its advantages, cloud computing still faces several limitations and challenges that must be addressed. Such challenges include scalability, data management, resource allocation, interoperability, and security and privacy [5]. Since cloud-based applications and frameworks often handle large datasets, effective resource management and scheduling are crucial for maintaining system performance.

Cloud computing frameworks such as Apache Spark [6], Dask [7], Knative [8], and others are designed to perform processing tasks on massive data sets efficiently, distributing these tasks across multiple computers, either locally or with the help of external tools. One widely used tool for managing cloud-based applications is Kubernetes [9]. Kubernetes simplifies the deployment, scaling and management of containerized applications [10]. Kubernetes' smallest deployment unit is called Pod.

Many complex applications, including Apache Spark, manage independent units of work (tasks) within the Kubernetes Pods [11]. This process introduces challenges because the

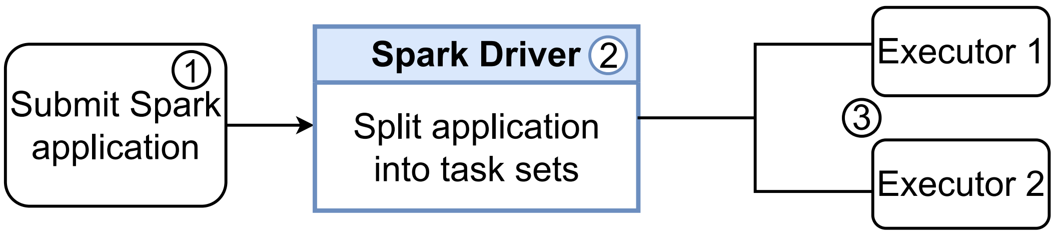

Figure 1.1: Spark application execution process.

Kubernetes scheduler is unaware of these tasks, leading to decreased performance and complicating the development of more complex scheduling algorithms.

1.1 Background

With the vast data growth, many new distributed computing frameworks have been developed, leading to the appearance of a subset of these frameworks — distributed application runtimes — that focus specifically on the runtime environment and execution of distributed applications [12]. These runtimes provide frameworks and tools for abstracting the complex underlying infrastructure. Examples of such distributed application runtimes include Apache Spark, Dask, and Knative, each offering unique features and capabilities.

This section introduces general concepts necessary for formulating research questions. Chapter 2 provides a more detailed description of the frameworks and internal workflows.

Apache Spark

Apache Spark [6] is a distributed computing framework designed to work with a cluster of computers, known as workers, to execute tasks. Spark manages the execution of a computational program or application by breaking it down into smaller, independently runnable tasks that are then executed on worker nodes. The general workflow is depicted in Figure 1.1. When a Spark application ① is submitted, a Spark Driver ② is created. The Spark Driver decomposes the application into smaller tasks, which are sent to worker nodes ③ for execution. The worker nodes run these tasks and report their execution status and completion to the Spark Driver.

Kubernetes

Kubernetes [9] is a powerful container orchestration platform for automating containerized applications' deployment, scaling, and management. It consists of a cluster of nodes,

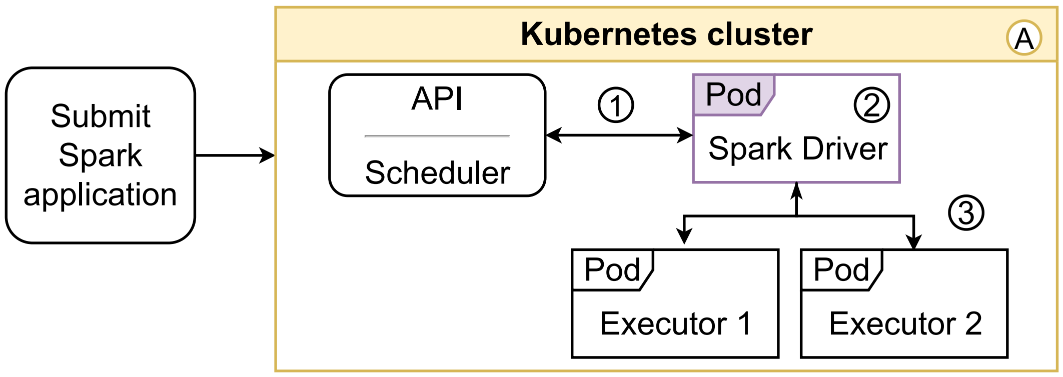

Figure 1.2: Spark application execution with Kubernetes.

including a controller node that manages the API server and scheduler, and worker nodes that run containers. Pods are the smallest deployable units in Kubernetes. The scheduler, managed by the controller node, allocates Pods to worker nodes based on CPU and memory utilization, node capacity and Pod affinity/anti-affinity rules [13].

In the context of Apache Spark, Kubernetes serves as a resource manager to simplify Spark's resource allocation and scaling capabilities. Spark applications are containerized into Docker containers and deployed as Pods within the Kubernetes cluster. While Kubernetes handles the scheduling of Pods, Apache Spark manages the execution of tasks within these Pods. Kubernetes is not aware of the individual tasks managed by Spark [11].

The Spark-Kubernetes workflow is illustrated in Figure 1.2. The Spark Driver ② is created as Pod within Kubernetes cluster Ⓐ and communicates with the Kubernetes API and scheduler ①. The Spark Driver performs the same steps as in a local Spark application environment, but also creates executor nodes ③ as Pods in the Kubernetes cluster.

Scheduler

A scheduler is an essential component of a distributed computing system. It makes dynamic decisions about task and data placement and directs tasks to different nodes in the cluster at the right time. The scheduler is defined by its architecture, mechanisms, and the actions it enables. The interaction of these components determines the overall system's performance [14]. Both Apache Spark and Kubernetes have their own schedulers, which operate at different levels and manage different resources within the system.

Kubernetes Operator

An Operator [15] in Kubernetes is a software extension that automates the management of applications and their components using custom resources. Several frameworks are available for creating and deploying Operators, offering a set of functional Kubernetes components and developer tools to simplify the creation of an Operator [16]. Examples of such frameworks include Operator Framework [16], Kubernetes Operator Pythonic Framework (Kopf) [17], and Metacontroller [18].

1.2 Problem Statement

Complex application frameworks like Apache Spark or Dask operate on large data sets. When an application is submitted to these frameworks, it describes the procedure that must be executed on the data. The framework then divides this application into smaller stages known as tasks, which are executed independently on worker nodes. Each framework has its own task scheduler that determines the order in which tasks are executed [19]. However, frameworks task-scheduling algorithms are relatively simple, which can result in suboptimal resource management and runtime performance.

To improve the scheduling workflow in these frameworks, for example, by implementing an energy-aware scheduler or a more complex scheduling algorithm, one needs to modify each local scheduler individually, as shown in Figure 1.3. This approach is time-consuming and inefficient. Instead, a unified scheduler outside the frameworks can be developed and applied across all frameworks with a single implementation.

This external scheduler could be integrated into the Kubernetes cluster environment and implemented as an Operator, as shown in Figure 1.4. In this setup, an external scheduler Ⓐ is introduced into the Kubernetes cluster, where it receives data from the Spark Driver through the Kubernetes API ①. The scheduler then creates executor nodes and allocates the tasks to these nodes ②. Upon completion of the tasks, the executors notify the Spark Driver of their status.

Currently, there are limitations in running Apache Spark with Kubernetes due to differences in their scheduling mechanisms. Kubernetes operates at the Pod level, while the schedulable units in Apache Spark are the tasks created within Pods. As a result, the Kubernetes scheduler is unaware of these tasks and cannot adjust the scheduling based on task-specific needs. Therefore, a novel scheduler that operates directly on tasks rather than Pods must address these limitations.

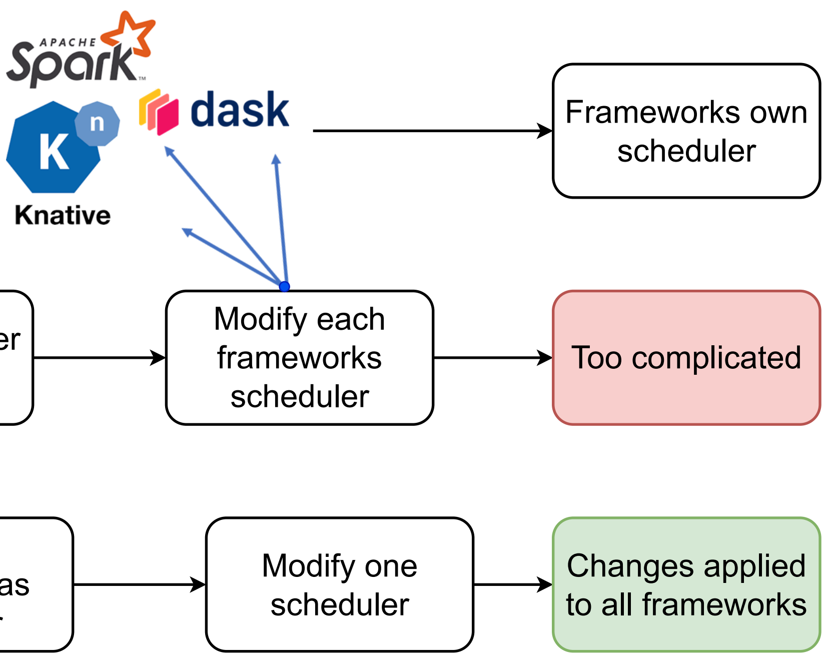

Figure 1.3: Illustration of the problem statement.

Several tools are available for creating a scheduler, and the choice of tool can significantly impact a system's runtime and efficiency. Since the end product system handles extensive data and requires low latency, selecting a suitable Operator framework is crucial for achieving optimal performance.

1.3 Research Questions and Methodology

To address this problem, we formulated the following general research question: **How to create a Kubernetes external scheduler that is aware of tasks generated by Spark Driver?** For this project, we have broken this general question down into three specific sub-questions:

R.Q.1: What is the comparative performance of the Operator Framework, Kopf, and Metacontroller in terms of the time taken to create a Pod in the Kubernetes environment?

An Operator is a software extension that allows developers to implement custom controller logic in Kubernetes. Various frameworks are available for creating Kubernetes Operators.

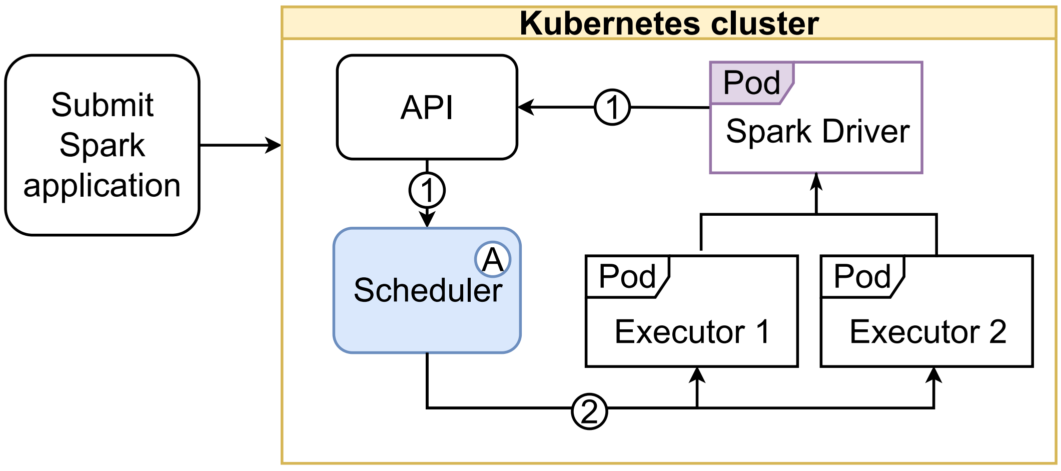

Figure 1.4: Spark application execution with external scheduler in Kubernetes cluster.

These frameworks operate on different programming languages, which can lead to inconsistencies in runtime performance and efficiency. Choosing a suitable framework is essential as the scheduler operates on large datasets and requires low latency for effective operation. However, there is a lack of comprehensive comparison across these frameworks, as no experiments have been conducted to evaluate their relative performances. This gap in prior research and measurements makes this research question particularly challenging.

Our methodology follows standard benchmarking practices in a field, specifically adapting the Graphalytics benchmarking methodology [20] for our context. Introduced in 2016, Graphalytics is a benchmark designed for graph analysis platforms, but its principles can be extended to other domains, including Kubernetes Operators [20]. To benchmark the performance of three well-known frameworks - Operator Framework (we use its component Operator SDK), Kopf, and Metacontroller - in terms of Pod creation, we design a similar benchmarking approach that defines essential components as workload, system, procedure, and goal.

The workload for our benchmarking is to create a Pod in a Kubernetes environment triggered by a Custom Resource (CR) Object. The process is as follows: we create a Custom Resource Object, the Operator detects CR Object creation, and instantiates a Pod. The system under test includes the Operator SDK, Kopf, and Metacontroller. Each of these frameworks' implementation is deployed in a Kubernetes cluster, with the cluster configuration maintained consistently across tests to ensure a fair comparison.

The benchmarking procedure involves deploying a Kubernetes cluster and each framework individually. For each framework, we execute the workload and measure the runtime and overhead. This benchmarking aims to compare the performance of different Kubernetes Operator frameworks based on the time taken to create a Pod. The specific objectives are to identify the framework with the lowest latency, assess the reliability of each framework, and provide insights into the trade-offs between different programming languages and frameworks in the context of Kubernetes Operators.

R.Q.2: How to design Task-aware scheduler compatible with Kubernetes Apache Spark stack?

Kubernetes is a container orchestration platform designed to automate the deployment, scaling, and management of containerised applications. In this context, a Task-aware scheduler is a component that manages individual tasks within a distributed application and operates externally to the central coordinator. For this research question, a compatible scheduler is one that can function well with Kubernetes and Apache Spark frameworks. Consideration of all these elements makes the design process challenging.

To answer this question, we aim to explore a Task-aware scheduler's architecture and working principles within Kubernetes, which involves designing and implementing a prototype solution by modifying the source code of the latest stable version of Apache Spark (v3.5.1). Modifications should enable data transmission to the Operator. Simultaneously, we need to develop custom logic for the Operator to receive and execute the data. We apply the atLarge design visions [21] to this novel design, creating a new solution for integrating a Task-aware scheduler with the Kubernetes Apache Spark stack.

R.Q.3: What is the impact on the performance of moving the scheduler outside of the Spark framework in the Kubernetes cluster?

This research question evaluates the performance implications of moving a Task-aware scheduler from the Apache Spark framework to an external component managed by Kubernetes. In contrast to the built-in scheduler of Apache Spark, specifically designed for distributed computing tasks, a Kubernetes scheduling component introduces a novel Task-aware scheduler.

To answer this research question, we use an existing benchmark framework to measure the runtimes of the original Spark system and compare them with the performance of the prototype scheduler developed in RQ2. This research question establishes whether the external Task-aware scheduler is as efficient (or at least performs comparably) as the existing Spark-Kubernetes system scheduler. Additionally, this analysis indicates whether further research and enhancement of the Task-aware scheduler are justified.

1.4 Thesis Contributions

This thesis's contributions can be grouped into three main areas: conceptual, technical and societal. The details of each contribution are presented below.

- **Conceptual:** findings of comparative analysis of the frameworks described in RQ1
- **Conceptual:** design of the external Task-aware scheduler
- **Technical:** implementation, deployment, execution and results collection of three frameworks for Operator creation
- **Technical:** implementation of the prototype of the external Task-aware scheduler
- **Technical:** execution of the experimental evaluation of the external Task-aware scheduler
- **Societal:** contribution to open-source project
- **Societal:** contribution to performance improvement in broad Kubernetes stack, partially related to Big Data
- **Societal:** satisfaction of grand social challenges listed in the Computer System manifesto [22]

These contributions together lead to a further publication.

1.5 Thesis Reading Guide

The first chapter briefly introduces the topic and problems solved within this project. Chapter 2 provides the background and related work information for all subsequent chapters.

The main body of the thesis consists of three chapters, each addressing one of the research questions. Chapter 3 compares the frameworks, answering RQ1. Chapter 4 provides a design of a Task-aware scheduler and answers RQ2. Chapter 5, answering RQ3, evaluates a system with the implemented prototype.

1.6 Plagiarism Declaration

I confirm that this thesis is my own work, has not been copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

2

Background and Related Work

This section provides an overview of three frameworks for creating Kubernetes Operators, highlighting their advantages and disadvantages. It also covers related research on scheduling methods for Spark and Kubernetes.

2.1 Kubernetes Operator Pythonic Framework (Kopf)

Kubernetes Operator Pythonic Framework (Kopf) is a Python-based framework and library designed to simplify the creation of Kubernetes Operators with minimal code [23]. Initially started as a Zalando [24] incubator project developed by Sergey Vasilyev, the project has been in maintenance mode since 2021. While it continues to receive necessary upgrades, no new features have been added.

Kopf is described as a simple yet powerful framework to use [23]. It is intuitive, supports all Kubernetes resources (CRs, built-in resources, etc.), offers various levels of handlers, and provides additional toolkits and integrations. Its comprehensive documentation makes it accessible to users at all levels [17].

The continuous popularity of Python, as shown in the TIOBE index for 2024 [25], further highlights the appeal of Kopf. According to TIOBE statistics [25], Python holds the first place with a 16.33% rating score, significantly higher than C (9.98%) or Golang (1.60%).

Python's widespread popularity implies that many developers are familiar with it and may prefer using it over learning a new language. Additionally, many books, tutorials, and other resources for learning Python make it highly attractive to beginners. Python's syntax does not require extensive additional studies, making it comparably easy to develop an Operator.

Table 2.1: Advantages and Disadvantages of Kopf

Advantages	**Disadvantages**
Simple and intuitive to use	Slower runtime compared to Go
Supports all Kubernetes resources	Limited to Python (has slower performance)
Comprehensive documentation	GIL limits concurrency
Popular language, many learning resources	Dynamic typing impact compilation speed
Minimal code needed to create Operators	Not ideal for performance-critical applications

Compared to languages like Go, a significant drawback of Python is its slower runtime. This distinction in runtime performance can be due to several factors, including typing style, execution mode and concurrency model [26]. Python is a dynamically typed language, meaning that the object type is determined at runtime. This dynamic typing can impact compilation speed, as Python needs to infer types during execution. In terms of execution mode, Python initially interprets bytecode and then compiles functions just-in-time. Additionally, Python's concurrency model is based on a global interpreter lock (GIL), which restricts only one thread from executing Python code at a time.

On the contrary, Go is compiled ahead-of-time, resulting in faster execution as the code is already compiled before runtime. It is also statically typed, allowing a program to compile faster as type declarations are determined at compile time. Moreover, Go utilizes its own Scheduler and provides user threads called *goroutimes*, allowing for more efficient concurrent execution without the limitations imposed by the GIL [27].

To conclude, while Kopf offers a straightforward and beginner-friendly framework for creating Operators in Kubernetes, it is essential to consider that Operators developed with Kopf may have slower runtimes compared to frameworks that utilize other languages like Go. This runtime difference can be significant, especially for performance-critical applications or those requiring high concurrency. Table 2.1 summarizes the advantages and disadvantages of Kopf.

2.2 Metacontroller

The closest equivalent to Kopf is Metacontroller. Like Kopf, Metacontroller aims to simplify the development of Kubernetes Operators without requiring profound knowledge of Kubernetes or complex programming languages [28]. Metacontroller began as a continuation of work on Google Kubernetes Engine (GKE) [29] and has since transferred into a community-maintained project [30]. Developed in 2017, Metacontroller is backed up by Google, the original developer of Kubernetes [28].

Unlike Kopf, where the core logic is written directly within the framework, Metacontroller supports two additional methods for handling the domain logic: Function-as-a-Service (FaaS) and In-cluster HTTP API deployments. Faas is a cloud-computing service that allows users to run functions (code snippets) in response to events without managing server infrastructure [31]. In-cluster HTTP API deployments involve deploying HTTP APIs that respond to the changes in CRs within the Kubernetes cluster. The primary logic of the Operator is contained within the FaaS functions or HTTP API deployments that execute actions based on the state of CRs [28].

A fundamental component in Metacontroller's architecture is the use of webhooks. A webhook is an HTTP-based callback function that allows one system to send real-time data to another based on the occurrence of specific events [32]. In Metacontroller, webhooks foster communication between the code logic and the Kubernetes API. Whether running as a FaaS function or an in-cluster HTTP API, the domain logic code acts as a webhook endpoint. The code and Kubernetes API interact by exchanging JSON [33] files. When changes in the CRs happen, Metacontroller sends an HTTP request to a webhook endpoint together with a JSON payload that describes the current state of the CR [18]. Since HTTP APIs are language agnostics (are independent of the programming language being used), Metacontroller allows domain logic to be written in any preferred programming language [28].

One potential disadvantage of Metacontroller is performance overhead. Webhooks and external HTTP endpoints can introduce latency, and the communication between Metacontroller and the webhook server may be significantly slower than operations within the Kubernetes cluster. Additionally, webhook frameworks have their own constraints, such as HTTP API timeout restrictions [28]. More issues may occur at a large-scale system as scaling and maintaining numerous webhooks can introduce additional operational overhead and complexity.

Table 2.2: Advantages and Disadvantages of Metacontroller

Advantages	**Disadvantages**
Supports multiple programming languages	Potential performance overhead due to webhooks
Allows domain logic to be written in any language	External HTTP endpoints can introduce latency
Supports Function-as-a-Service (FaaS)	Complexity in large-scale systems
Flexible architecture using webhooks	HTTP API timeout restrictions
Community-maintained project	Additional operational overhead for scaling and maintenance

In summary, Metacontroller provides a flexible way to create Kubernetes Operators using any programming language developer prefers. However, this flexibility has a trade-off in performance and potential complexity, especially in large-scale environments. Table 2.2 depicts the advantages and disadvantages of the Metacontroller framework.

2.3 Operator SDK

In contrast to Kopf and Metacontroller, Operator SDK is not a framework - it is a Software Development Kit (SDK), which generates the skeleton code for an Operator that users can enhance with the required domain logic [28]. - an open-source toolkit designed to manage Kubernetes Operators automatically and at scale [34]. However, it is considered a complicated tool for writing Operators [28]. Additionally, both Operator SDK and Operator Framework only support the Go programming language.

Go (or Golang) [35] was developed by Google in 2009 for constructing backend software services. From the beginning, it was designed to perform at a comparable level with languages like C and C++ while maintaining a relatively easy syntax similar to Python or JavaScript [36]. As a compiled language, Go offers faster code execution. It also has built-in support for concurrency, allowing it to handle multiple tasks simultaneously. The main advantages of Go include speed, concurrency and simplicity. However, Go is still gaining popularity and does not have as many study resources as Python. Additionally, it has some unique concepts like *goroutines* that developers need to familiarize themselves with [37]. Overall, Go is a powerful and relatively simple language with some learning curves.

Operator SDK provides the necessary tools to build, test and package Operators. It initially helps users integrate their application business logic with the Kubernetes API, allows for application improvement, and provides a user experience of cloud services. Operator SDK uses the controller-runtime libraries (a set of Go libraries for building Controllers [38]) that simplify writing Operators by providing high-level APIs and abstractions, tools for code generation and various extensions to cover the most common Operator use cases [34].

Additionally, Kubernetes is written in Go, making the interaction of Go and Kubernetes API robust. Using Operator SDK with Go provides a more complex yet more substantial implementation, with no limitations on the functionality of a custom Operator [39]. Developing an Operator with Go is the most popular among these options, as 71% of Operators on OperatorHub [40] (Kubernetes community to share Operators) are written in Go.

Despite its strengths, Operator SDK has several limitations. Firstly, it supports only the Go programming language for writing Operator logic, requiring developers to be familiar with Go. Secondly, it is more complex to use and learn than Kopf or Metacontroller, as it requires a solid understanding of Kubernetes concepts, APIs, and Operator concepts like Manifests. Lastly, Operator SDK is not generally supported on Windows, producing an additional challenge for developers unfamiliar with Linux. While there are options to use a virtual machine or Windows Subsystem for Linux (WSL) to install Operator SDK on Windows, there are no specific interfaces for Windows, meaning that development would occur primarily via the Linux command-line interface.

To conclude, Operator SDK is one of the most powerful but also one of the most complex toolkits for building Kubernetes operators. It requires a good understanding of Kubernetes, Operators, and the Go language, which makes it less friendly for beginners. However, its flexibility and robustness make it a substantial preference for experienced developers looking to create highly functional and scalable Operators. The advantages and disadvantages of Operator SDK are shown in Table 2.3

2.4 Local Spark Schedulers

The imperfections of the Apache Spark scheduling system have led to the development of new custom schedulers. These custom schedulers generally fall into two categories: those that enhance the existing Spark stand-alone scheduler and those that integrate with Kubernetes but do not go beyond the Pod level. In this section, we discuss enhancements

Table 2.3: Advantages and Disadvantages of Operator SDK

Advantages	**Disadvantages**
Provides tools to build, test, and package Operators	Requires familiarity with Go language
Supports multiple development options (Go, Ansible, Helm)	More complex to use and learn than Kopf or Metacontroller
High flexibility and robustness with Go	Not generally supported on Windows
Built-in support for Kubernetes API	Requires a solid understanding of Kubernetes concepts
Widely used and supported by the community	Development primarily via Linux command-line interface

to local Spark schedulers, and in the next section, we cover custom schedulers built on Kubernetes.

One example of a local scheduler is RUPAM. Proposed in 2018, Rupam is a heterogeneity-aware task scheduler whose primary goal is to improve performance by considering task-level resource characteristics and underlying hardware characteristics preserving data locality [41]. While being successful and improving the performance up to 62.3% compared to the standard Spark scheduler, RUPAM enhances the existing local scheduling technique but does not address the issue of scheduling tasks within the Kubernetes cluster.

Numerous other research publications such as [42] and [43] propose novel Job scheduling algorithms and approaches like greedy Best-Fit-Decreasing (BFD) algorithm [42], dynamic adaptive scheduling approach [43] and others. They follow a similar goal of improving the existent performance of Apache Spark, reaching it from different perspectives. For instance, [42] research aims to reduce the cost of Virtual Machines usage in the cloud environment, whereas [43] research approaches the trade-off problem between latency and throughput in the batch-based streaming systems. Job scheduling is a prevalent topic in the Spark scheduling research area. However, similar to RUPAM, these researches target only built-in Spark schedulers and do not address the problem of scheduling the Spark-Kubernetes ecosystem.

2.5 Kubernetes Custom Schedulers

Custom Kubernetes schedulers like Volcano [44] or Yunikorn [45] are widely used when running Spark on Kubernetes. Volcano is a Kubernetes Native Batch System for deploying high-performance workloads on Kubernetes. It is the only scheduling project of Cloud Native Computing Foundation (CNCF) [46] that is designed to provide support for diverse scheduling algorithms, multi-architecture computing, and mainstream computing frameworks leading to more efficient scheduling [44].

YuniKorn is a universal resource scheduler developed initially for YARN [47] and Kubernetes [48]. The main goal of YuniKorn is to unify the existing schedulers in Kubernetes, such as the default scheduler for services and scheduler extensions like Kube-batch [49] for batch scheduling, as well as YARN's Capacity Scheduler and Fair Scheduler for batch workloads [48].

Both schedulers provide some advanced scheduling capabilities like gang scheduling [50] or support of various workflows and fine-grained scheduling of resources [51]. However, these schedulers are built on top of Kubernetes and thus cannot operate on the levels that Kubernetes cannot manage, namely beyond Pods.

3

Operator Framework Comparison

This chapter answers RQ1 - *What is the comparative performance of the Operator Framework, Kopf, and Metacontroller in terms of the time taken to create a Pod in the Kubernetes environment?*

The previous chapter described each chosen framework and provided background information. Here, we explain the experiment design, setup, and evaluation of the results. Appendix A contains the hardware and software specifications, executable commands, and their order.

3.1 Overview

As of 2022, Kubernetes was the leading container orchestration tool for running workloads in containers at scale [52]. Kubernetes simplifies the process of running applications and plays a crucial role in their design, deployment, and management, making these processes more straightforward and ensuring high availability for users.

The user interacts with Kubernetes via the command line interface tool *kubectl* that sends requests to the Kube-apiserver [13]. The Kube-apiserver, in turn, manages all the components on the node and extracts all the necessary data from the key-value storage Etcd. The user manages the resources Kubernetes has (whenever such communication occurs) through the *kubectl* or Kubernetes Dashboard (a graphical user interface alternative to *kubectl*). There are various types of resources, such as Pods (the smallest unit in Kubernetes) or ReplicatSets (the exact copy of the Pod) [13]. Also, Kubernetes supports Custom Resources (CRs) that enable users to create and interact with the resources they require in the same way as with the built-in Kubernetes resources. Such CRs are frequently used with Operators - software extensions of Kubernetes that allow new functionality to be

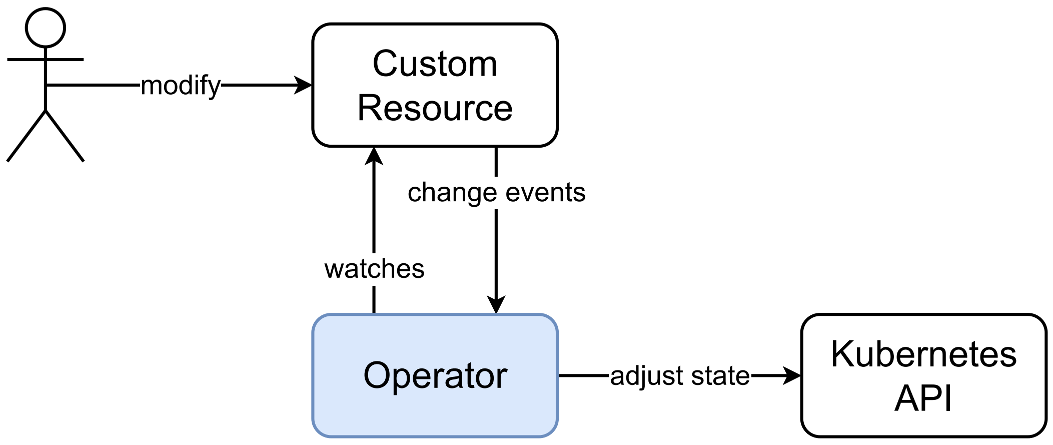

Figure 3.1: the process of interaction of an Operator with CRs and Kubernetes API [39].

added. Operators act as controllers, constantly checking the status of the CRs and taking actions to align the current state with the desired state specified within the CR [13]. The process is illustrated in Figure 3.1.

There are multiple frameworks for creating Operators for Kubernetes, each supporting different languages and implementation approaches. Some of these frameworks are Operator SDK [16], Kubernetes Operator Pythonic Framework (Kopf) [17], and Metacontroller [18].

Despite the large number of libraries and tools available for creating Operators for Kubernetes, a thorough comparison of these frameworks has not yet been conducted, posing challenges in selecting the most suitable framework. While some blog posts like [39] have attempted to compare various frameworks based on simplicity of implementation, limitations, and popularity, they lack experimental analysis of runtime differences among these frameworks.

A comprehensive comparison of various frameworks benefits developers who want to select the right tool based on their specific requirements. It also lays the foundation for future research to identify the most efficient framework, encouraging the development of robust and scalable Kubernetes solutions.

3.2 Design Requirements for Operator Frameworks

This section describes the requirements for the Operator design. These requirements ensure that the design is aligned with the objectives and constraints and provide a foundation

Table 3.1: Requirements for Operator creation framework

ID	Requirement
R1	The Operator must be efficient and handle a significant volume of tasks.
R2	The programming language used for the Operator should have high runtime performance
R3	The framework should be straightforward, allowing developers to create Operators with minimal effort.
R4	The chosen framework should provide a balance between runtime speed and implementation simplicity

for evaluating the performance and success of the Operator. Table 3.1 provides the requirements for the Operator's design.

This paper aims to prototype a Task-aware scheduler built as an Operator. In this experiment, the scheduler is deployed within a single cluster, managing a limited number of tasks in parallel. However, in the real-world scenario, the scheduler must handle a significant volume of tasks concurrently. Therefore, the Operator's runtime efficiency is crucial (**R1**). The choice of framework can significantly impact runtime performance, influenced by factors such as the programming language used. For instance, Python [53] tends to have slower runtimes [26] compared to other languages like Go [35] or PHP [54] (**R2**).

Furthermore, an Operator framework should be relatively easy to implement, ensuring a lower entry barrier for beginners (**R3**). An ideal framework would be a trade-off between runtime speed and implementation simplicity, enabling the creation of scalable and efficient Operators with minimal development effort (**R4**).

3.3 Design of Framework Comparison Experiment

The experiment described in this chapter aims to compare the runtimes and simplicity of implementation of various frameworks and toolkits that help create Kubernetes Operators. No such comparison has been made before, so this research aims to provide a more informed and reasonable choice of frameworks for Kubernetes Operators.

Although many frameworks are available, this research focuses on Kopf, Metacontroller and Operator SDK. We chose frameworks based on their popularity, simplicity of implementation, programming language support, and available documentation. Kopf is considered to be the most straightforward framework for implementing an Operator. However, it only supports Python, which might affect its runtime performance. Metacontroller, while slightly more complex to implement, allows developers to choose any preferred programming language. For this experiment, Metacontroller is implemented with both Python and Go. Operator SDK with Go is the most complex toolkit, but its use of Go might significantly enhance runtime performance due to Go's efficiency.

Alternative frameworks considered included Operator SDK with Ansible or Helm, which are easier to build than Operator SDK with Go but do not offer flexible functionality. Another alternative is Side8's k8s-operator, the direct equivalent of Kopf, which functions well only with scripts written in shell or bash [28]. The most complicated approach is writing an Operator using a bare programming language like Java or Kotlin. However, this method requires a deep and detailed understanding of Kubernetes and Operators [39].

In this experiment, Kopf, Metacontroller, and Operator SDK with Go are independent variables, while the comparison metrics are dependent variables. The quantitative metrics include the mean, median, and interquartile range (IQR) of the runtimes of each framework. The general distribution of runtime frequencies is visualized using violin plots. The definitions of the objects are identical for each framework to ensure a fair comparison. Similarly, the core logic implemented is kept consistent and adjusted only for the specific programming language needs.

The runtime of each framework is tested under two conditions: with and without Pod creation. Each framework is coded to create a new Pod when the YAML file of the Custom Resource Object is created inside the Kubernetes cluster. To measure the overhead, the core logic is modified to output a print statement without making a Pod. We repeat both procedures 100 times and save each iteration's runtimes (milliseconds) in an external CSV file. We repeat this process for each framework and collect all results in a single file used for the analysis. As a control benchmark, we also measure the runtimes of the Kubernetes built-in Pod creator.

Overhead is an essential measure describing the additional time frameworks require to create a Pod compared to running the program without making a Pod. Overhead directly impacts a framework's overall performance and efficiency. When comparing multiple frameworks, overhead is a critical factor. It helps select the framework that performs well and has minimal additional costs associated with auxiliary tasks. In our evaluation, overhead is computed as follows:

$$\text{Overhead} = \text{Mean runtime with Pod} - \text{Mean runtime without Pod}$$

This design and evaluation approach leads to a comprehensive and objective decision regarding which toolkit better addresses this project's needs. We additionally compare the custom Operator to a built-in Kubernetes object creator, providing a benchmark to understand how the custom Operator performs relative to native Kubernetes functionalities.

Each framework is assessed based on the availability and quality of documentation and tutorials to measure implementation simplicity. Additionally, the subjective experiences of a developer unfamiliar with these frameworks (the author) are considered. The developer reports on the ease of understanding and applying the frameworks, noting any challenges. Both objective resources and subjective experiences are combined to evaluate implementation simplicity comprehensively.

We perform this experiment in both local and cluster environments for more comprehensive results. This approach allows us to evaluate the performance and scalability of our solution under different conditions. By comparing the outcomes from these environments, we can gain insights into potential bottlenecks and optimize the scheduler for various deployment scenarios.

This experiment provides a detailed and fair comparison of Kopf, Metacontroller, and Operator SDK by evaluating their runtimes and implementation simplicity. The results offer valuable insights into each framework's performance and efficiency, aiding in selecting the most suitable tool for creating an external Task-aware scheduler.

3.4 Setup for the Framework Comparison Experiment

We run the experiment on a local machine and a cluster. The environment specifications and framework installation procedures are listed in Appendix A. Note that the local machine used is significantly less powerful, directly affecting the results.

Core logic

In all frameworks, we implement a core logic that creates a Pod whenever a Custom Resource Object is created. The Custom Resource Definition (CRD) and Controller files (used only for Metacontroller and Operator SDK) are created manually for Kopf and Metacontroller; Operator SDK creates them automatically. These files and the CR objects are written as YAML files (a human-friendly data serialization language for all programming

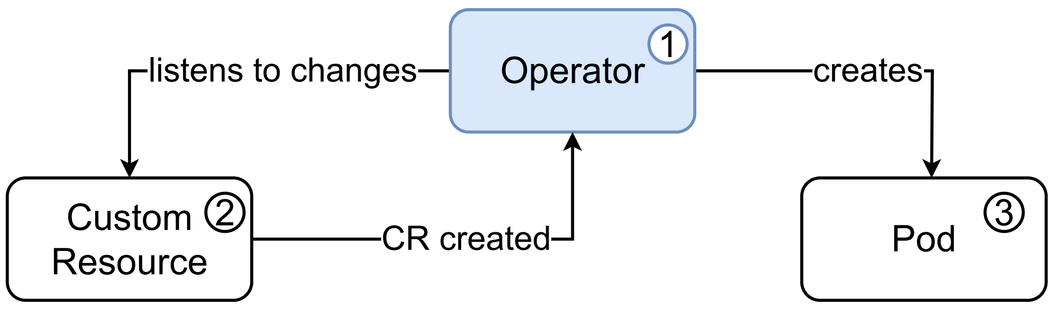

Figure 3.2: Workflow of an Operator.

languages [55]). CRD, Controller and Object files are created using *kubectl apply* command. Using the *kubectl create* command is also possible, but it does not maintain the changes applied to a resource [52].

An Operator's workflow is illustrated in Figure 3.2. A new CR Object ② can be created once CRD and Controller files are applied to the Kubernetes cluster. This Object has its own name. An Operator ① is coded to watch for changes in the Objects of a newly defined type (CR) and react by creating a new Pod ③ with the name specified as the name of the CR Object. An example of such an Object is:

```
apiVersion: example.com/v1
kind: MyPod
metadata:
  name: mypod-2
```

This is an example of an Object used for the Metacontroller framework. It is of the MyPod kind (defined using CRD) and has the name *mypod-2* (*mypod-1* is used with Kopf, *mypod-3* with Operator SDK). Once this Object is created, the code constructs a Pod named *mypod-2* respectively.

In general, the procedure can be described as follows:

1. create CRD within the cluster
2. create Controller within the cluster (if needed)
3. run the core logic code in another terminal (runs endlessly before a keyboard interruption to watch for changes in the CR objects)
4. create a CR Object within the cluster
5. check if a new Pod has been created

This procedure applies to Kopf and Metacontroller; steps 1 - 3 differ for Operator SDK. The workflow for each framework is described in more detail in Appendix A.

To measure the runtimes, we create a PowerShell program that starts a timer when CRD and Controller files are created, and the core logic code is running. The program measures how long it takes to run the *kubectl apply* command, then saves this runtime in milliseconds. The timer only counts the time it takes to create an Object, not to delete it. After the 100 iterations, all runtimes are saved in a CSV file.

We execute these procedures on a local device and then rerun all the same steps in the cluster. We analyze the obtained data using Python pandas, seaborn, and matplotlib.pyplot libraries. We visualize the frequency of runtimes using violin plots for three frameworks (two plots for Metacontroller). We also calculate the mean, median, and interquartile range (IQR) in Python.

3.5 Results of the Framework Comparison Experiment

This section provides the results of the runtime comparison between three different frameworks. The data collected from 100 runs for each framework (both with and without Pod creation) is summarized to detect performance differences.

We visualize the runtimes distribution using violin plots sorted by median values in ascending order. For the violin plots, the X-axis represents experimental frameworks - Kopf, Metacontroller (Meta and GO_Meta), Operator SDK, and a control variable (Kubernetes Pod creator labelled k8s). The Y-axis represents the runtimes in milliseconds. Each violin plot is bifurcated to display runtimes with and without Pod creation, with accompanying box plots indicating quartile ends and medians.

We evaluate the metrics' mean (the average runtime), median and interquartile range (IQR) for a more constructive analysis.

We discuss the results obtained by running the experiment locally and on the cluster.

Local environment

The comparison results are depicted in Figure 3.3. The Figure shows that all frameworks outperform the Kubernetes built-in runtime for Pod creation. This observation suggests that frameworks likely employ more efficient resource allocation and management strategies or implement better concurrency and parallelism mechanisms tailored to specific tasks, thus exceeding the more general-purpose Kubernetes mechanisms. Among the frameworks,

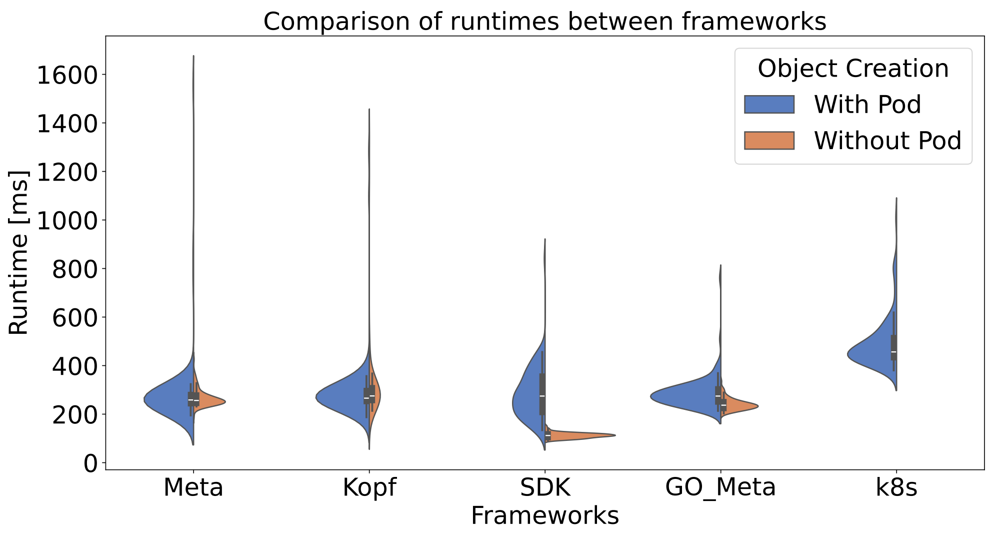

Figure 3.3: Local runtimes of frameworks ordered based on median values.

Descriptive statistics of different frameworks with and without Pod creation running locally

Table 3.3: With Pod creation

Framework	Median	Mean	IQR
Kopf	265.82	292.87	43.79
Meta	258.44	286.06	36.59
GO_Meta	274.50	287.13	52.71
SDK	274.50	284.18	148.50
k8s	456.43	494.05	81.44

Table 3.4: Without Pod creation

Framework	Median	Mean	IQR
Kopf	275.04	307.52	51.49
Meta	256.34	267.31	33.28
GO_Meta	236.85	240.48	26.87
SDK	112.00	111.73	14.50
k8s	-	-	-

Metacontroller with Python has the fastest runtime, followed by Kopf. While SDK and Metacontroller with Go perform slower, they exhibit fewer outliers than Meta and Kopf.

The analysis further delves into quantitative metrics such as mean, median, and interquartile range (IQR), summarized in Tables 3.7 and 3.4. Table 3.4 does not include Kubernetes (k8s) values because we used a built-in Kubernetes command to create the Pod, which cannot be modified to output a print statement.

Metrics Evaluation

Based on the results, the SDK framework demonstrates the best overall performance with a mean runtime of 284.18 ms, followed closely by Meta (286.06 ms) and GO_Meta (287.13

Table 3.5: Overhead of the frameworks

Framework	With Pod	Without Pod	Overhead
Kopf	292.87	307.52	-14.65
Meta	286.06	267.31	18.75
GO_Meta	287.13	240.48	46.65
SDK	284.18	111.73	172.45

ms). Kopf exhibits slightly slower performance with a mean runtime of 292.87 ms, but the difference is minimal. However, it's important to note that the mean is sensitive to outliers, and all frameworks exhibit some extreme values, as depicted in Figure 3.3.

Contrary to the mean, the median offers a more robust representation of the typical runtime, particularly in cases where data is skewed or contains outliers. In this context, Meta outperforms all other frameworks with a median runtime of 258.44 ms. Kopf follows with a median runtime of 265.82 ms, while both GO_Meta and SDK have a median runtime of 274.50 ms. Notably, the median value of the Kubernetes built-in Pod creator (k8s) is significantly slower compared to the other frameworks.

Similarly to the median, the IQR provides a measure of variability that is robust to outliers, focusing on the central portion of the data. A smaller IQR indicates more consistent runtimes. Meta demonstrates the smallest IQR at 36.59 ms, indicating highly consistent runtimes. Kopf follows closely with an IQR of 43.79 ms, while GO_Meta exhibits an IQR of 52.71 ms. Surprisingly, the IQR of SDK is almost four times larger than that of Meta, with an IQR of 148.50 ms, surpassing the Kubernetes built-in Pod creator (k8s) with an IQR of 81.44 ms.

The overhead of the frameworks is presented in Table 3.5.

In the evaluation of framework overhead, Kopf displays a negative overhead of -14.65 ms, indicating that it takes less time to create a Pod than to execute a print statement. This characteristic is common in Python due to its interpreted nature. Metacontroller demonstrates a modest overhead of 18.75 ms, while GO_Meta's overhead is slightly higher at 46.65 ms. Conversely, the SDK framework demonstrates the most significant overhead among the evaluated frameworks, with a value of 172.45 ms.

Cluster Environment

Running the experiment within the cluster produced slightly different results, presented in Figure 3.4.

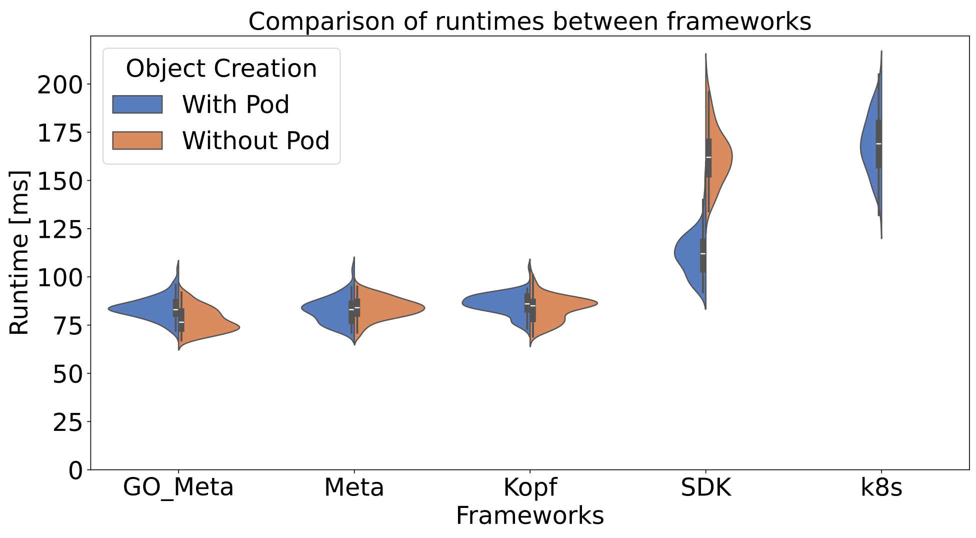

Figure 3.4: Cluster runtimes of frameworks ordered based on median values.

Descriptive statistics of different frameworks with and without Pod creation running on cluster

Table 3.7: With Pod creation

Framework	Median	Mean	IQR
GO_Meta	76.5	77.68	9.25
Kopf	85.0	83.48	9.25
Meta	84.0	83.79	6.50
SDK	162.0	163.00	17.25
k8s	169.0	168.98	22.25

Table 3.8: Without Pod creation

Framework	Median	Mean	IQR
GO_Meta	83.0	83.81	6.25
Kopf	86.0	85.85	7.00
Meta	83.0	82.69	9.25
SDK	112.0	111.73	14.50
k8s	-	-	-

According to the violin plots, the Metacontroller and Kopf frameworks perform significantly better than the Operator SDK and Kubernetes' built-in Pod creator. Surprisingly, Metacontroller with Go runtime is dramatically faster on the cluster than on the local machine. The scope of the runtime values is smaller on the cluster, and the frameworks have few outliers. The distribution of the runtimes is more normal.

Metrics Evaluation

The metrics information is summarized in Table 3.8.

Based on the results, GO_Meta stands out as the most efficient framework with the lowest mean runtime of 77.68 ms, closely followed by Kopf with a mean runtime of 83.48

Table 3.9: Overhead of the frameworks

Framework	With Pod	Without Pod	Overhead
GO_Meta	77.68	83.81	-6.13
Kopf	83.48	85.85	-2.37
Meta	83.79	82.69	1.1
SDK	163.00	111.73	51.27

ms and Meta at 83.79 ms. In contrast, SDK and k8s show significantly slower performance, with SDK's mean runtime being 163.00 ms and k8s - 168.98 ms.

While the mean runtime offers a general performance overview, it is sensitive to extreme values and may not always accurately reflect typical performance. Even though cluster results have very few outliers compared to local results, they might still affect the performance and are important for analysis. However, based on average runtime values, GO_Meta is the fastest, followed by Kopf and Meta, with Kopf being 0.31 ms faster than Meta. Operator SDK and k8s have slower average runtime values.

The median runtime offers a more robust representation of typical performance as it minimizes the impact of outliers. Here, GO_Meta again has the lowest median runtime of 76.5 ms. Meta and Kopf follow with 84 ms and 85 ms, respectively. SDK and k8s show significantly higher median runtimes, with SDK at 162.0 ms and k8s at 169.0 ms.

The interquartile range (IQR) provides insights into consistency. GO_Meta demonstrates the smallest IQR of 9.25 ms with Pod creation, indicating highly consistent performance with minimal variability. Kopf and Meta follow with IQR values of 9.25 ms and 6.50 ms, respectively, with Pod creation showing slightly higher variability than GO_Meta. Interestingly, SDK exhibits the largest IQR of 17.25 ms with Pod creation, suggesting higher variability in its performance. Similarly, the IQR for k8s is quite large at 22.25 ms, reflecting more significant variability and less consistent performance.

The overhead is presented in Table 3.9.

Table 3.9 shows that GO_Meta and Kopf have negative overheads. For GO_Meta, this might be due to efficient resource management and the benefits of the Go language's concurrency model. The negative overhead for Kopf, which also appears in the local experiment, is likely due to Python being the interpreted language. Metacontroller has a very low overhead of 1.1 ms, but Operator SDK shows the most significant overhead among the frameworks, with a value of 51.27 ms. This substantial overhead indicates that creating a Pod introduces additional expenses, possibly due to less efficient resource allocation or higher overheads associated with its containerization process.

Implemetation Simplicity

In assessing the ease of implementation for each framework, particularly from a beginner's perspective, Kopf appears to be notably user-friendly. Its comprehensive documentation and straightforward setup process promote a smooth experience, with installation taking less than an hour and simple Operator creation being complete in under three hours. Clear instructions and detailed online resources with numerous examples significantly ease the learning curve.

In contrast, the Operator SDK framework presents several challenges, with less accessible documentation and a setup process requiring a deeper understanding of unfamiliar concepts. Implementation with WSL (Operator SDK is not supported on Windows) adds complexity, leading to longer implementation times and multiple troubleshooting sessions, potentially challenging for users with limited experience.

Metacontroller framework is in a balance between the two, offering moderately clear documentation and a manageable setup process. Although the initial setup of Metacontroller can be complex, the availability of tutorials provides a smoother implementation compared to the Operator SDK framework.

3.6 Analysis of Results from the Framework Comparison Experiment

The experiment evaluated the performances and usability of several Kubernetes frameworks, including Metacontroller (Python and Go implementations), Kopf, and the Operator SDK. The experiment was executed on a local machine and in the cluster environment. Since the cluster environment has more computational power and resources than the local machine, the results obtained from the cluster are generally more indicative of each framework's performance capabilities. Thus, the conclusions are drawn based on cluster results.

The analysis reveals that Metacontroller, implemented in Go, is the fastest framework overall. This efficiency can be attributed to Go's optimized concurrency model and effective resource management, as demonstrated by its negative overhead. Following GO_Meta, Kopf and Metacontroller with Python show comparable runtime speed, with Kopf having a slightly lower mean runtime than Meta. Despite this, both frameworks demonstrate strong performance and consistency.

In contrast, Operator SDK demonstrates a significantly slower runtime. SDK's substantial overhead highlights additional costs associated with Pod creation and reveals a more considerable variability than the ones of other frameworks.

From a usability perspective, Kopf appears to be the most user-friendly framework, with a straightforward setup process and extensive documentation. Metacontroller balances ease of use and technical depth, providing a manageable learning curve and helpful resources, but the setup and implementation with Go is more complex than with Python. In contrast, Operator SDK presents the most significant challenges for newcomers, with more complex documentation and a more challenging setup process, especially for users with limited experience.

Overall, GO_Meta is recommended when the best performance and efficiency are required, while Kopf is ideal for users seeking ease of implementation and good performance.

3.7 Summary

In this chapter, we answer RQ1 by experimentally comparing three frameworks used to create Operators in Kubernetes. The Metacontroller implemented with Go and Kopf appears to be the most suitable and efficient framework. While the Metacontroller provides faster runtime, implementing with Kopf is more straightforward, requires less time, and is slightly less efficient than Metacontroller with Go. Thus, we use the Kopf framework in this project to develop a Task-aware scheduler.

At the end of this chapter, we evaluate the Kopf framework according to the design requirements set before. Table 3.10 summarizes the requirements and provides their statuses: yes (requirement is passed), partially (requirement is passed partially) and no (requirement is not passed). As seen in Table 3.10, the chosen framework satisfies almost all of the requirements except for **R2**. While Kopf has high runtime performance, there are other frameworks like Metacontroller with Go that have more efficient runtimes. Hence, **R2** is satisfied partially.

Table 3.10: Evaluation of requirements for Operator creation framework

ID	Requirement	Satisfied
R1	The Operator must be efficient and handle a significant volume of tasks	Yes
R2	The programming language used for the Operator should have high runtime performance	Partially
R3	The framework should be straightforward, allowing developers to create Operators with minimal effort	Yes
R4	The chosen framework should provide a balance between runtime speed and implementation simplicity	Yes

4

Task-Aware Scheduler Design

In this chapter, we answer RQ2 - *How to design Task-aware scheduler compatible with Kubernetes Apache Spark stack?* by designing an external Task-aware scheduler that runs inside the Kubernetes cluster.

4.1 Overview

With the rapid growth of data, big data processing has become essential across various fields, including healthcare, banking, scientific research, and large and small businesses. These sectors rely heavily on advanced data processing frameworks, with Apache Spark being one of the most prominent tools available [56].

Apache Spark [6] is an open-source framework for processing large datasets with a straightforward API [57]. It supports multiple programming languages, including Scala and Python, and is widely used for data engineering, data science and machine learning. Apache Spark has an extensive ecosystem with integrations into various machine learning, analytics, and storage frameworks [6]. Users can also customize Apache Spark to meet specific requirements by modifying the Spark source code, available on GitHub, along with installation and building guides [58].

4.2 Problem With Local Frameworks Schedulers

Apache Spark and similar Big Data frameworks operate on significant data volumes. To understand how Spark manages these data-processing tasks, it is helpful to look at the process through which a Spark application is decomposed into manageable components. A schematic workflow is illustrated in Figure 4.1. In this workflow, the Spark application (1)

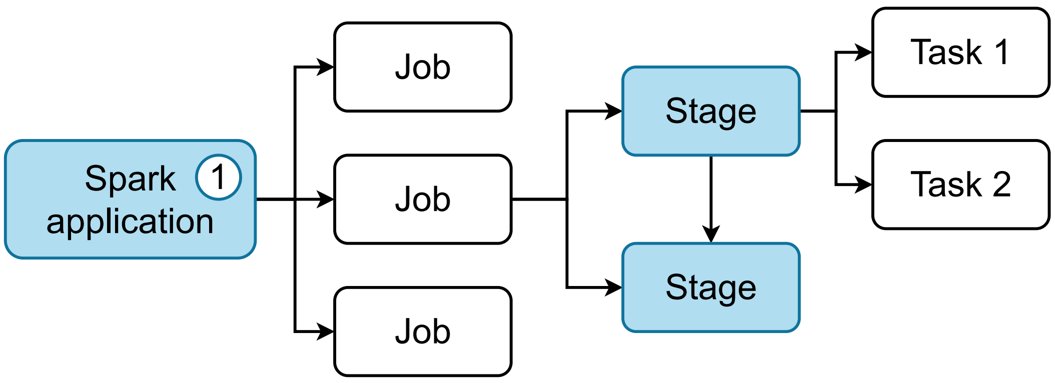

Figure 4.1: Decomposition of Spark application to executable tasks [59].

represents the computational program that performs various operations across a cluster in parallel [61].

When an application is submitted, a central coordinator called Spark Driver is created. The Spark Driver is responsible for breaking down the application into smaller components called Jobs. These Jobs are further divided into stages, which are then split into tasks, as shown in Figure 4.1. These tasks can be executed independently across worker nodes on local resources or through external systems like Kubernetes that make application management more accessible [9]. In a Kubernetes environment, the smallest unit of deployment and management is called a Pod [62]. When Spark runs with Kubernetes support, the Spark Driver runs as a Pod within the Kubernetes Cluster [60], as shown in Figure 4.2. The problem is that Kubernetes is only aware of and can manage the Pods at a high level without direct visibility into the internal processes of the Spark Driver, making the scheduling process less efficient than it might have been.

One potential solution to improve scheduling efficiency in Spark applications running on Kubernetes is to create an external scheduler within the Kubernetes cluster. This external scheduler receives application data from Spark Driver, applies a scheduling algorithm, and executes operations on the worker nodes. This way, Kubernetes can adjust its scheduling strategies based on the tasks instead of Pods, leading to more efficient scheduling.

Additionally, having an external scheduler centralizes the scheduling logic into a single program, making assessing, modifying, and enhancing the scheduling algorithms easier. Instead of working with the complex and distributed Spark's built-in scheduling mechanisms, developers can experiment with more sophisticated scheduling algorithms (moving beyond the default First-In-First-Out (FIFO) scheduling) outside the Spark source code.

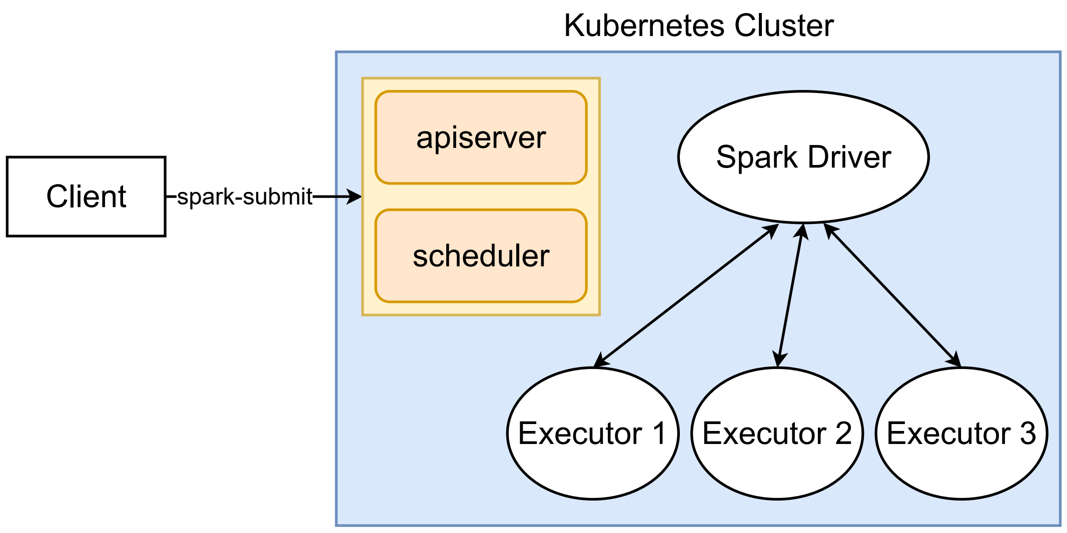

Figure 4.2: The mechanism of submitting a Spark application to Kubernetes cluster using spark-submit command [60].

4.3 Design Requirements for Task-Aware Scheduler

In this section, we define the requirements for an external Task-aware scheduler. These requirements are presented in Table 4.1 and are evaluated at the end of the next chapter because these requirements apply to both RQ2 and RQ3.

The scheduler must be functional (**R1, R2, R3**), work with many tasks (**R5**), and work well with the existing ecosystem of Spark and Kubernetes (**R4, R6**). It should implement more efficient scheduling algorithms (**R7**) and provide monitoring tools (**R8**). The scheduler should also be compatible with multiple Spark versions (**R9**).

4.4 Design Of Task-Aware Scheduler

The primary purpose of this study is to develop an external Task-aware scheduler within the Kubernetes cluster to enhance the management of Apache Spark, Dask, and other similar application frameworks. This external scheduler functions as a united scheduling solution, allowing the integration of advanced and innovative scheduling techniques, such as energy-aware scheduling, without requiring a complete redesign for each individual framework.

Figure 4.3 provides a high-level overview of the design of this external scheduler. When an application is submitted using the spark-submit command, the Spark Driver is created and initializes communication with the Kubernetes Operator through the Kubernetes API.

Table 4.1: Requirements for an external Task-Aware Scheduler

ID	Requirement
R1	The scheduler must store information about executors
R2	The scheduler must decide which executor to use and notify Spark Driver about the decision
R3	The scheduler must be notified when the task assigned to an executor is completed
R4	The scheduler must be comparable to at least with the built-in Spark scheduler
R5	The scheduler must efficiently manage a high volume of Spark tasks
R6	The scheduler should integrate with Spark and Kubernetes
R7	The scheduler must support the implementation of advanced scheduling algorithms beyond default FIFO
R8	The scheduler should offer monitoring and logging capabilities for task execution
R9	The scheduler should be compatible with multiple Spark versions and configurations

The Operator processes the data received from the Spark Driver and executes it on the worker nodes. As the tasks are completed, the executor nodes notify the Spark Driver of their progress and status.

Research Methodology

This project adopts the design and development methodology, which refers to developing a novel system or function for a specific situation. In particular, this project utilizes the Bottom-Up approach, which starts with a foundation and builds up towards a solution [63]. All the modifications are produced incrementally, from the most minor and simplest modifications to more complex and structured ones.

This project follows applied research principles. Applied research is a non-systematic way (meaning that its primary goal is to find a solution immediately) to solve a specific research problem. Out of three types of applied research, this project uses research and development, which focuses on creating or designing new products or services that address the specific needs of particular markets in society [64].

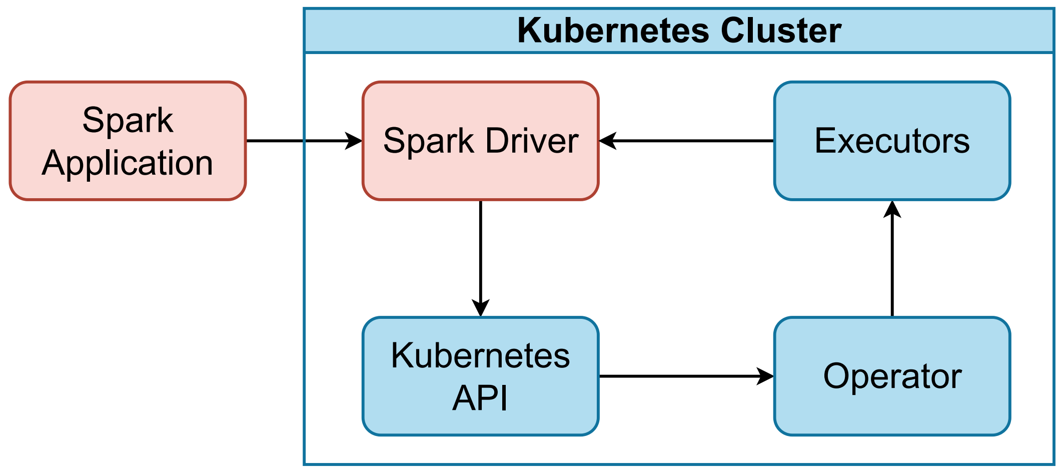

Figure 4.3: The overview of the Task-aware external scheduler.

Current Spark Scheduler Workflow

Scheduling in Apache Spark happens in various stages and locations. An overview of the scheduling is shown in Figure 4.4, and a more detailed execution workflow is presented in Figure 4.5.

After submitting the Spark application as a user program (1), the resilient distributed datasets (RDDs) (2) are created for every application action. RDD represent work that can proceed independently on one executor node. Spark action is an RDD operation that triggers the RDD transformation and returns the results. Each action triggers a new Job creation. Each Job has a separate Direct Acyclic Graph (DAG) (3) with relevant RDD objects (named Stages) as graph nodes. The Direct Acyclic Graph (DAG) is subsequently used as an input for the DAG Scheduler (4). DAG Scheduler comes up with an execution plan to run all the Stages listed in the DAG with minimal time. DAG Scheduler divides the DAG into sets of tasks (named TaskSets) that can be executed in parallel.

The TaskSets (5) created by the DAG Scheduler are submitted to a low-level TaskScheduler (6). TaskScheduler schedules and launches the tasks via cluster managers. First, TaskSetManager is created to manage the TaskSet, sorting the tasks by priority, using their Job ID, and then by Stage ID. Then, the TaskSetManager is added to the pool. The pool has a tree-like structure, which depends on the Scheduling Mode. Scheduling Mode in Spark only supports the default First-In-First-Out (FIFO) and FAIR (scheduling method where all jobs get an equal share of resources over time [65]). If the Scheduling Mode is FIFO, there is only a single root pool. In the case of FAIR, there can be multiple pools,

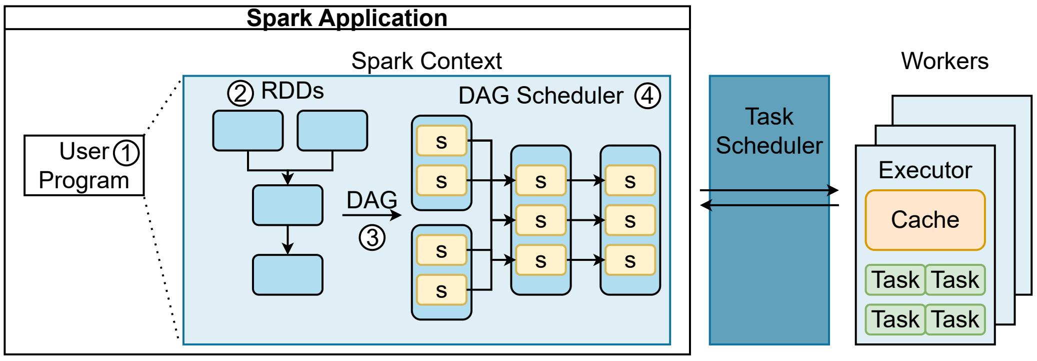

Figure 4.4: Overview of Spark scheduling [19].

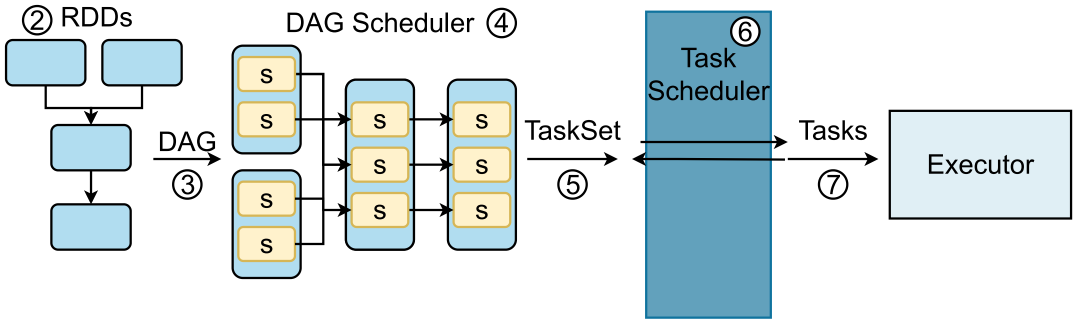

Figure 4.5: The execution workflow of Spark scheduling [19].

which can also be nested [66].

A backend scheduler (a subclass of SchedulerBackend trait, which provides a unified interface for cluster environments to interact with TaskSchduler) sends the information of available resources in each executor it manages to the TaskScheduler. If the TaskScheduler returns a positive decision, the SchedulerBackend assigns the tasks ⑦ from the pool to the executors [67].

Since the scheduling process happens on multiple layers, several options exist for implementing it externally. The best case is to separate the entire scheduling process from the top-level DAG Scheduler, as shown in Figure 4.8 Ⓐ. This case is the most beneficial as it allows the implementation of a different scheduling algorithm for the entire process. However, this approach is also the most complex as it requires implementing the scheduling structure from scratch, including all the bottom layers and backup procedures.

Another approach is to implement the task set scheduling process, represented in Ⓑ. The DAG can be sent to the custom Operator, which handles the TaskSetManager and Pool creation and implements the task-scheduling algorithm. Such a method still allows some

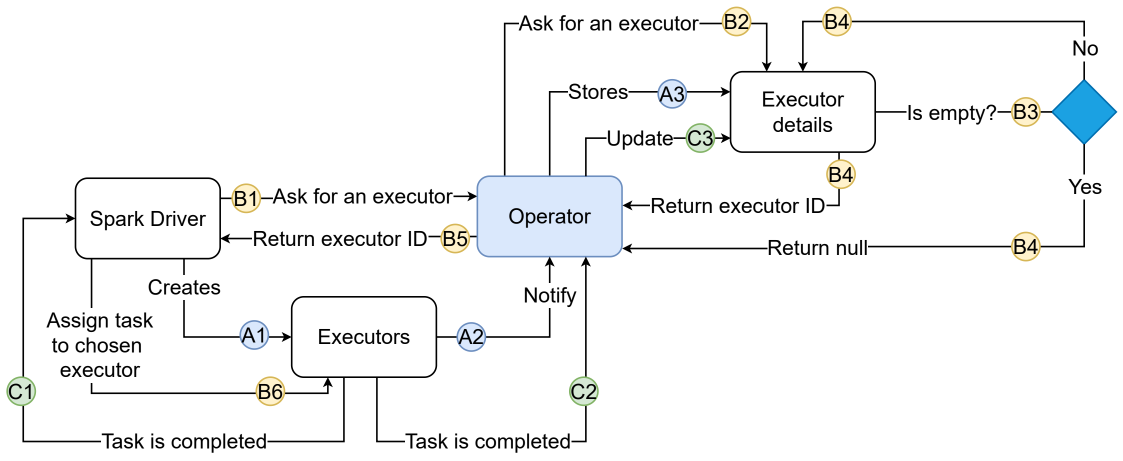

Figure 4.6: Operator workflow for executing the tasks.

freedom in choosing the scheduling algorithm; however, it is bounded by some limitations as it has to follow the already created DAG.

The most straightforward implementation is manipulating tasks from the TaskSet as depicted in (C). The tasks are sent to the executors in the order they arrive from Spark Driver, following the FIFO principle of the original Spark Scheduler. This approach does not introduce any novel scheduling algorithm, but it proves that extracting the task-scheduling process outside of the Spark Driver is possible.

Due to time constraints, the latter approach was implemented in this project.

4.5 Implementation Of Task-Aware Scheduler

This project describes a Task-aware scheduler that aims to solve the problem of the Kubernetes scheduler being unaware of the tasks derived in Spark Driver. We approach this by creating an external scheduler that handles communication between Spark Driver and executors, sending the task data to the chosen executor nodes. This external scheduler can be implemented as an Operator in Kubernetes using the Kopf framework.

When Spark Driver decides which executor to submit the task to, a custom Operator can make this decision instead. Communication between Spark Driver and Operator can be done using the Kubernetes client Fabric8 [68] by creating a new Custom Resource Object (CRD should be defined and applied to Kubernetes beforehand). Figure 4.6 describes the workflow process as a flow chart. Communication between Spark Driver and Operator happens in several cases:

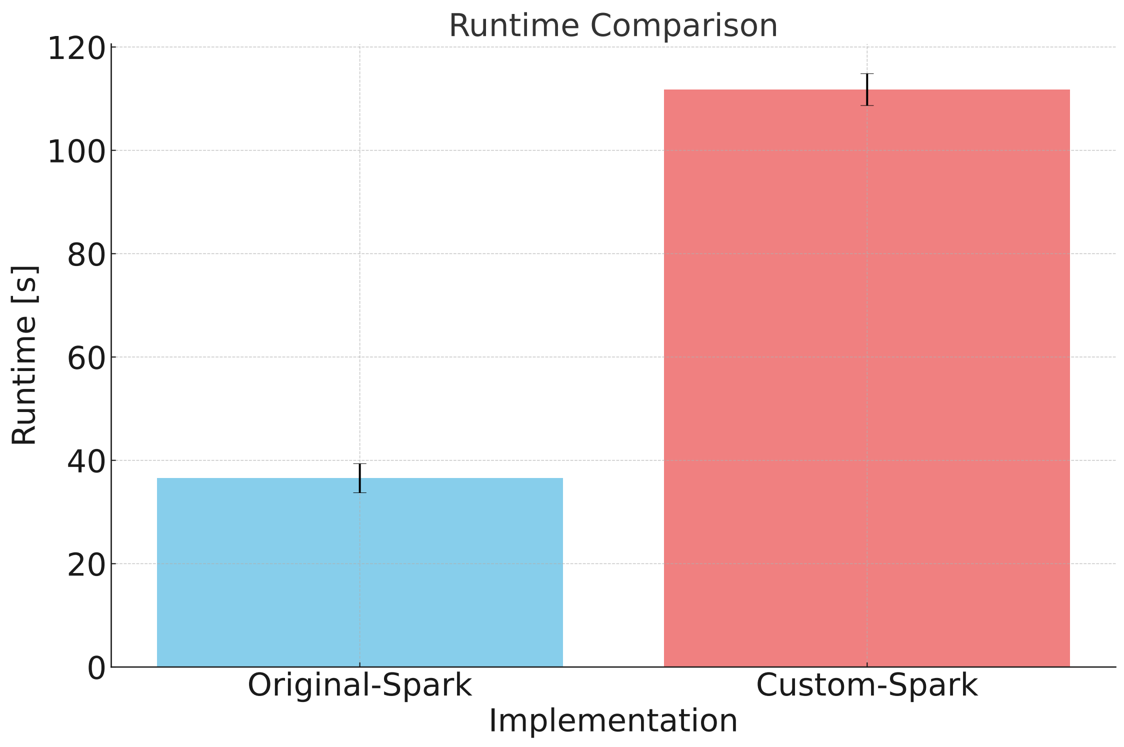

Figure 4.7: Comparison of original and custom Spark runtimes.

- New executor is created (A1) - information about the new executor, including executor ID, hostname and the number of tasks running on an executor, is specified as a spec field of spark-executors CR (A2). Information about each executor is stored on an Operator (A3).

- Spark Driver asks which executor to choose for the task (B1) - Spark Driver creates a spark-request CR with an empty field for an executor ID. The operator (B2) checks if there are any executors created (B3) and decides which executor has the lowest number of running tasks (B4). It then updates the empty field of the spark-request CR with the corresponding executor ID (B5). Spark Driver reacts to this change by using the chosen executor (B6).

- Task is assigned to an executor - when assigning a task to an executor, the corresponding spec field *number of tasks running* is updated. The Operator receives this update and reflects the changes in the data structure that stores all information about the executors C3.

- Task is completed - when the task has been completed on an executor, it notifies Spark Driver C1 and Operator C2 about this. The operator updates the *number of tasks running* of that particular executor C3.

One of the requirements for a Task-aware scheduler is to develop an efficient solution, meaning that the custom Spark runtime working with a new scheduler should have performance comparable to the original Spark runtime. To assess this, we test the performance of the modified Spark (with our novel component) by running the SparkPi example and measuring its execution time. The results, shown in Figure 4.7, indicate that the system with the new scheduler performs approximately three times slower than the original one. While this is a noticeable difference, it is not excessively or dramatically slower than the original Spark runtime.

Based on these results, we can conclude that the design of the external Task-aware Scheduler is comparable to the original Spark scheduler in terms of performance. However, it is essential to note that this evaluation is limited. The SparkPi example only consists of two tasks, which is too simplistic to draw comprehensive conclusions about the scheduler's performance. Therefore, a more thorough evaluation of the system, which considers a broader range of applications and scenarios, is presented in the next chapter.

Limitations

This research aims to show that it is possible to create an external scheduler that is aware of the tasks generated by Spark Driver. Therefore, the implementation only focuses on the limited functionality, ignoring all additional background stages implemented by the original Spark scheduler. Thus, the proposed Task-aware scheduler is a restricted version of the one Spark initially has and is expected to perform worse in runtime.

Additionally, the Spark used was the last stable model, and all the dependencies were adjusted to that particular Spark version. Because this project does not utilize the latest Kubernetes, Fabric8, and Spark versions, it might have some limitations that were updated in the newest releases. For future releases of Spark, updating these dependencies would be required.

4.6 Summary

In this work, we designed and implemented an external Task-aware scheduler that communicates with the Spark Driver and executes tasks on the Kubernetes executors. The scheduling algorithm here is FIFO, the same as the default Spark algorithm.

This section answered RQ2 by providing a working solution. Proving the possibility of creating such an external scheduler is good groundwork for developing more complex Task-aware schedulers. In our view, having an external scheduler is beneficial for the general performance of Spark applications and is a solid foundation for future research.

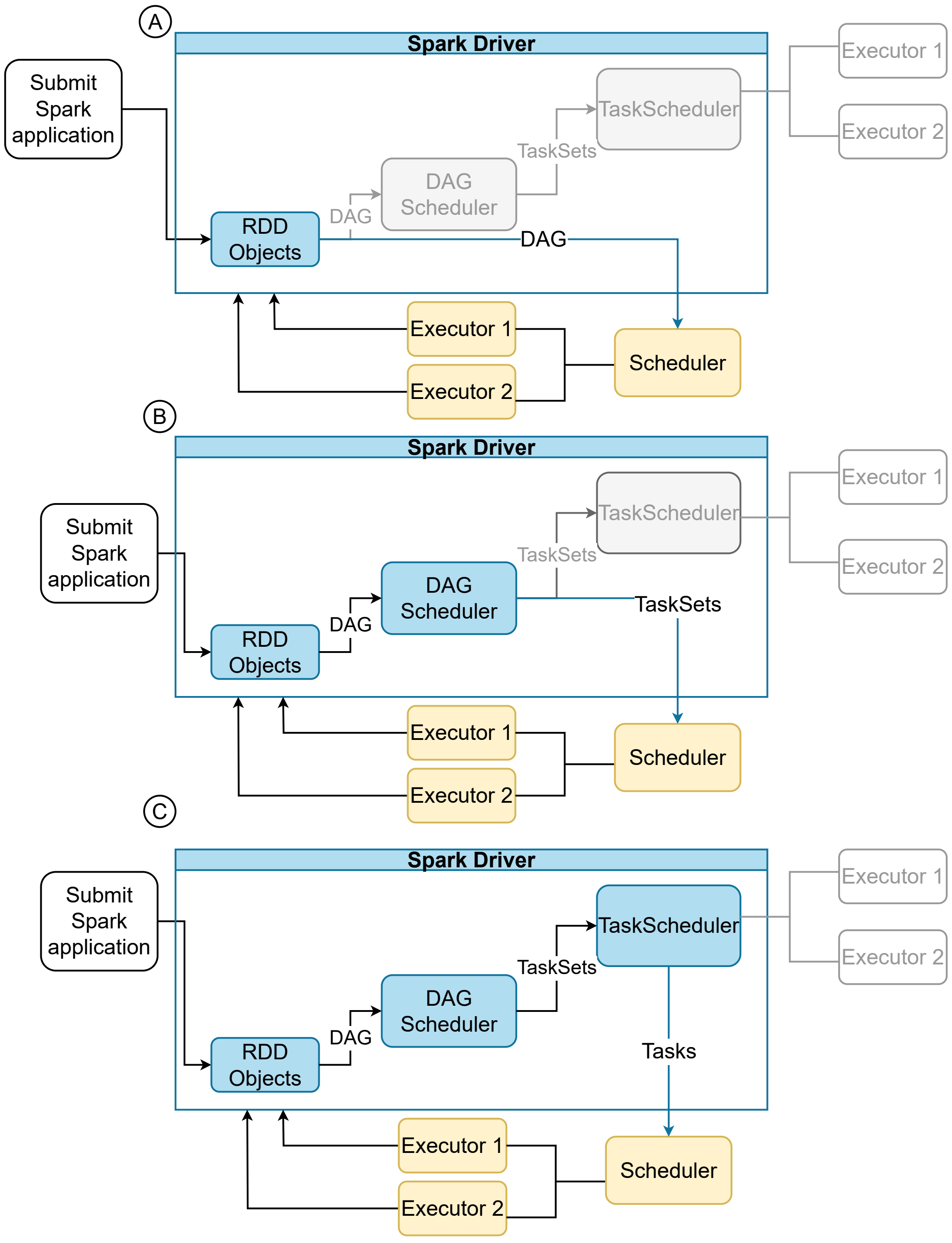

Figure 4.8: Three options for an external scheduler.

5

Experimental Evaluation

This section aims to analyze the performance of the runtime framework with an added external scheduler and answer RQ3 - *What is the impact on the performance of moving the scheduler outside of the Spark framework in the Kubernetes cluster?* by running experiments in a test environment.

5.1 Experimental Setup

To evaluate the performance of the Task-aware scheduler, we run an application on a cluster to determine whether the runtime of the custom system is comparable to that of the original Apache Spark runtime. Computations were performed on an AL01 cluster node six operated by atLarge group. For this evaluation, we used the Continuum [69] deployment and benchmarking framework, along with the Sharebench framework. The specifications of the virtual machines used for this evaluation are detailed in Table 5.1.

We used Continuum to generate and build an emulated system. Continuum is a benchmarking framework for the edge-cloud compute continuum. Its main goal is to develop a comprehensive framework and tools for managing and optimizing the performance of large-scale distributed systems. Continuum emulates a compute continuum environment by automating the creation of a cluster of cloud, edge, and endpoint virtual machines. Framework automatically installs operating services, resource managers (including Kubernetes), and applications inside the emulated cluster [69].

In this evaluation, the Continuum runs virtual machines with five cloud nodes, one of which is a controller node. Continuum also starts a Kubernetes cluster and verifies whether all nodes in the cluster are connected.

Table 5.1: Virtual machines specifications.

Provider	Nodes	Core	Memory
qemu	5	4	32 GB

We generated 10 GB of data and ran tests using the Sharebench [70] framework. Sharebench is a novel benchmarking framework introduced by Lennart Schulz. It is an infrastructure framework for automated performance analysis of distributed resource-sharing mechanisms. In this evaluation, the Sharebench framework generates data and metadata for SQL queries and runs TPC-DS. TPC-DS is a Decision Support Benchmark model for queries, data maintenance, and other aspects of a decision support system. TPC-DS provides a representative performance evaluation as a decision support system [71].

To evaluate the performance of an external Task-aware scheduler, we compared the runtime of 20 TPC-DS queries running with the original Spark and our modified version. Running custom Spark added additional challenges as it requires some source code modifications of the Sharebench framework, which Appendix A addresses.

5.2 Findings

The evaluation results are depicted in Figure 5.1 as two logarithmic scale box plots demonstrating the distribution of runtimes. The results have been visualized using a box plot with a logarithmic scale for better understanding. The X-axis represents the Spark version (original and custom), and the Y-axis shows the runtime in seconds. We compared two Spark versions - the original Spark (left box plot) and the custom Spark (right box plot).

Box plots show that the modified Spark is significantly slower and less consistent than the original Spark. The custom Spark's median runtime is larger than the original Spark's, indicating a slower average performance. Additionally, the custom Spark has a wider IQR, suggesting that the middle 50% of runtimes are less consistently grouped around the median compared to the original Spark. The whiskers for the custom Spark are also longer, indicating more significant variability in runtimes. While there is at least one outlier for the original Spark, the overall spread of data points is more extensive for the custom Spark, showing greater variability. In summary, the custom Spark demonstrates longer and less reliable runtimes, whereas the original Spark shows a lower median runtime with decreased performance variability.

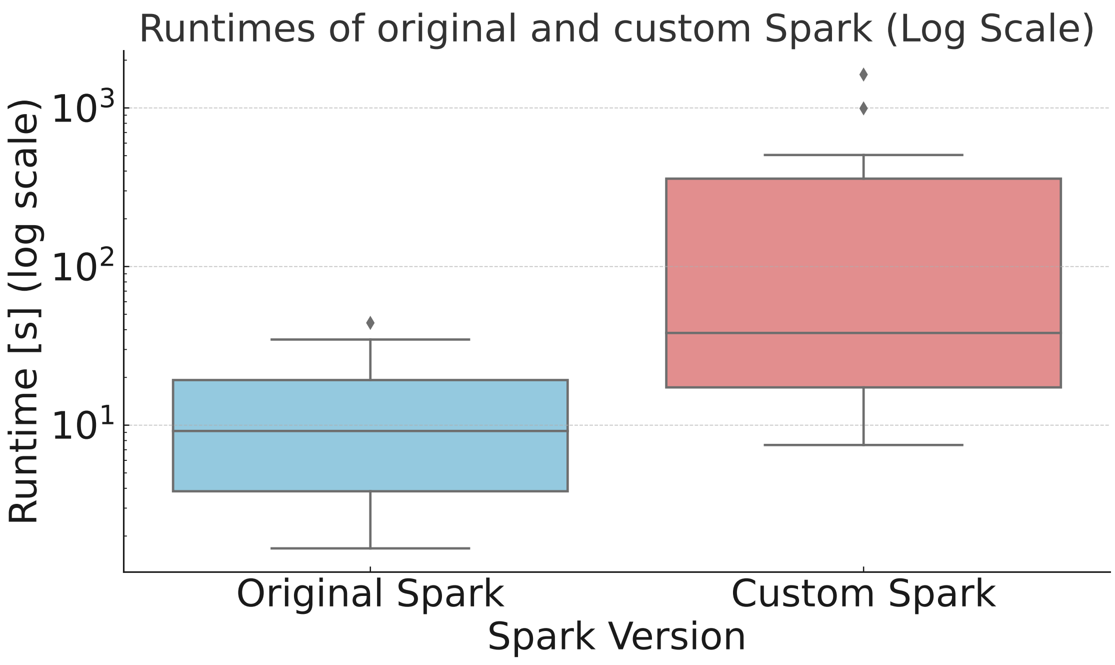

Figure 5.1: Runtimes of original and custom Spark.

Table 5.2: Metrics of original and custom Spark in seconds.

Spark version	Mean	Median	IQR
Original Spark	12.85	9.19	15.15
Custom Spark	273.49	38.14	340.89

Statistical metrics like mean, median, and IQR values are summarized in Table 5.2, which provides more quantitative analysis. The average runtime (mean) for the original Spark is 12.85 s, while for the custom Spark, it is 273.49 s. These statistics show that, on average, the custom Spark takes significantly longer to run. Custom Spark is 21.3 times slower, on average, than the original. The median runtime for the original Spark is 9.19 s, compared to 38.14 s for the custom Spark. A higher median in the custom Spark indicates that the central value of the custom Spark runtimes is higher, meaning that more than half of the tasks run faster with the original Spark than with the custom Spark. The IQR for the original Spark is 15.15 s, while for the custom Spark, it is 340.89 s. The larger IQR for the custom Spark indicates greater variability in runtimes, meaning the custom Spark version's performance is less predictable.

The box plots and the table show that the custom Spark appears to have a less consistent

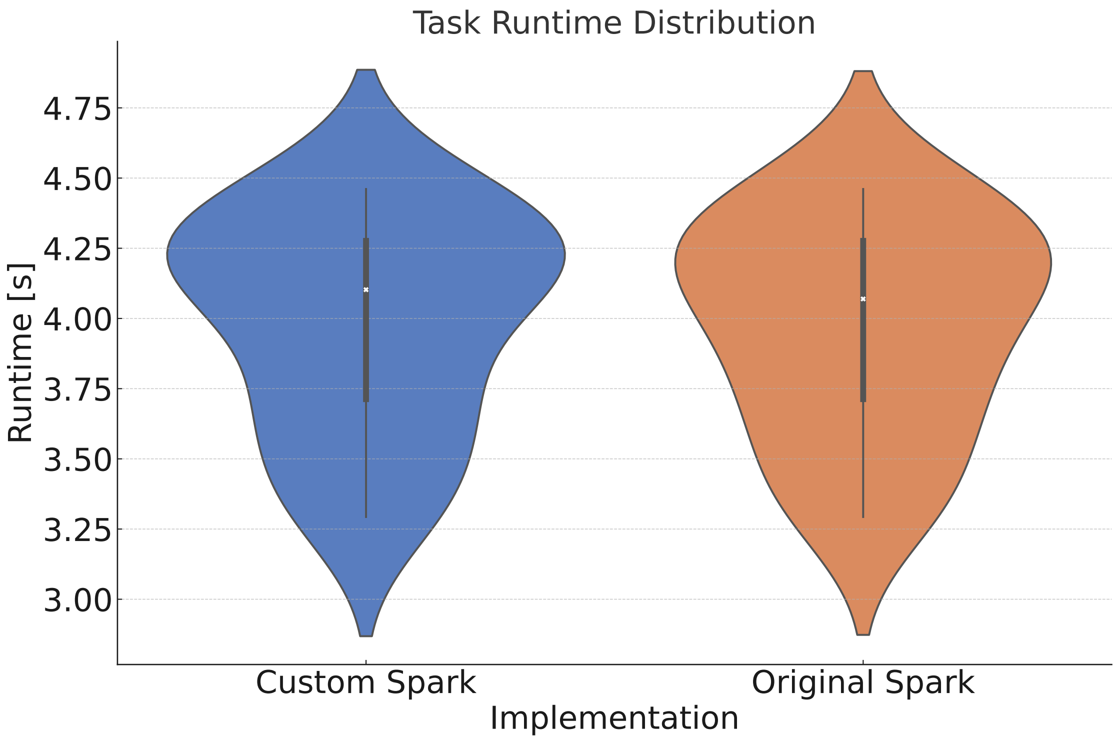

Figure 5.2: Task execution times of original and custom Spark.

and predictable performance than the original Spark. The custom Spark has a higher average and median runtime, indicating that it performs worse than the original Spark in terms of speed. Custom Spark's wider distribution and higher IQR also suggest that its performance can vary significantly depending on the specific task or conditions, whereas the original Spark delivers more stable performance.

To evaluate whether the Task-aware scheduler increases the task execution runtime, we collected the runtimes of individual tasks within the Spark application of both implementations. These runtimes are presented as violin plots in Figure 5.2. The X-axis represents the implementation, and the Y-axis shows the runtime in seconds. From this Figure, we can see that both implementations have similar performance in terms of task runtimes. Both distributions have the same overall shape and close median runtime values, which suggests a similar variability and central tendency for the task runtimes of both implementations. The original Spark shows a slightly more spread-out distribution towards the lower end of the runtime range, suggesting more variability in faster runtimes. The custom Spark reveals a similar distribution spread but is slightly less variable in faster runtimes.

In general, custom Spark appears to be significantly less efficient and reliable. This result is expected because an external scheduler in custom Spark supports only basic functionalities and introduces additional communication, negatively affecting the runtime. For practical implications, the original Spark is more suitable at the moment. Regarding task execution, both implementations are relatively consistent and comparable in their task runtime performance.

5.3 Limitations

Although this analysis provides representative results for the experimental evaluation, several limitations remain.

Data

The generated data has a sample size of 10 GB, which is sufficient for the current evaluation scale. However, using a larger dataset of 1 TB or more could provide more robust and comprehensive insights into the performance of the Spark versions.

Another important consideration is the randomness of the queries. In this evaluation, we run 20 queries for the original Spark and another 20 for the custom Spark version. These queries are not identical but are randomly generated each time, which means that running the same experiment with a different set of random queries could yield different results. For a more consistent comparison, identical queries should be used across different runs. Additionally, the generated SQL queries vary in size, significantly impacting their runtimes.

Environmet

The evaluation is conducted on a cluster with virtual machines, where the number of nodes and memory for the Kubernetes cluster are predefined. Changing these specifications, such as using different node counts, memory configurations, or even different types of virtual machines, could lead to other performance results.

Moreover, replicating this experiment on a local machine or another cluster could yield different outcomes due to hardware, network conditions, and system load variations. The results obtained in our specific environment might not be fully generalized to other settings.

Evaluation metrics

The primary metric used for evaluation is the runtime of the two Spark versions. While this is a crucial measure of efficiency, it does not address other important aspects such

Table 5.3: Requirements for an external Task-Aware Scheduler

ID	Requirement	Satisfied
R1	The scheduler must store information about executors	Yes
R2	The scheduler must decide which executor to use and notify Spark Driver about the decision	Yes
R3	The scheduler must be notified when the task assigned to an executor is completed	Yes
R4	The scheduler must be comparable to at least with the built-in Spark scheduler	Yes
R5	The scheduler must efficiently manage a high volume of Spark tasks	Partially
R6	The scheduler should integrate with Spark and Kubernetes	Yes
R7	The scheduler must support the implementation of advanced scheduling algorithms beyond default FIFO	No
R8	The scheduler should offer monitoring and logging capabilities for task execution	Yes
R9	The scheduler should be compatible with multiple Spark versions and configurations	Partially

as resource utilization (CPU, memory), scalability, fault tolerance, and overall system stability.

For instance, resource efficiency can be critical in determining the cost-effectiveness of each Spark version, especially in large-scale deployments. To provide a more holistic performance evaluation, metrics such as CPU usage, memory consumption, and disk I/O should be considered. Additionally, evaluating how each Spark version scales with increasing data sizes and workloads and how it handles failures and maintains stability under stress would offer valuable insights.

5.4 Evaluation of Task-Aware Scheduler Requirements

In this section, we evaluate the requirements identified at the beginning of Chapter 4. The statuses of the requirements include yes (for a passed requirement), no (for the requirements that are not passed), and partially (for the partially passed requirements, respectively). The requirements are shown in Table 5.3.

Due to limited time availability, we implemented a basic version of the Task-aware scheduler without providing the functionality for dynamic task allocation and advanced scheduling algorithms. Hence, **R7** is not satisfied. Based on the evaluation results, our scheduler handles a high volume of Spark tasks but is unable to manage them efficiently, thus satisfying the **R5** only partially. Also, currently, our scheduler only works with Spark v3.5.1, leading to only partial satisfaction of **R10**.

5.5 Summary

In this chapter, we answered RQ3 about the performance of modified Spark by conducting an experimental evaluation of the modified version of Spark with an added Task-aware scheduler.

The results show that Spark's performance changed negatively after moving the scheduler outside of the framework. The original Spark version is more efficient and reliable. The modified Spark version shows higher variability and slower average runtimes, is less predictable, and needs further optimization.

To address the limitations of the evaluation process, future work should use larger datasets with identical queries across runs, experiment with different environmental setups, and include a broader set of evaluation metrics to provide a more comprehensive performance evaluation.

6 Conclusion

This section summarizes the key aspects of the thesis by answering research questions, discussing the limitations, and providing an intuition for future work.

6.1 Answering Research Questions

This thesis addresses the scheduling problem of the Apache Spark application within the Kubernetes cluster. To enhance scheduling and resource management efficiency, we propose creating an external Kubernetes scheduler that is aware of the tasks generated by Spark Driver. We limited this problem to three research questions and were answering them within this thesis project.

RQ1 - What is the comparative performance of the Operator Framework, Kopf, and Metacontroller in terms of the time taken to create a Pod in the Kubernetes environment?

To answer this RQ, we selected three frameworks for Operator creation, designed and implemented an example workflow and executed it with each framework locally and on the cluster.

The results indicate that the most efficient framework in terms of runtime is the Metacontroller implemented with Go. However, Kopf and the Metacontroller framework implemented with Python show very close results to Metacontroller with Go and also have efficient runtimes. Regarding the ease of implementation, Kopf is the dominant representative.

Based on the results and analysis, we used the Kopf framework to create an external Task-aware scheduler. Thus, RQ1 is answered by providing and analysing the results and choosing the framework for RQ2.

RQ2 - How to design a Task-aware scheduler compatible with Kubernetes Apache Spark stack?

To answer this RQ, we used design processes typical for the field, created a novel design solution, and implemented a prototype. We modified the Spark source code and created a scheduler as a Kubernetes Operator. To establish whether the novel Task-aware scheduler is comparable to Spark's existing scheduler, we tested the original and modified Spark versions on a basic example (SparkPi). We then compared the runtimes and confirmed that our scheduler is indeed comparable.

This RQ is answered by providing a working design (presented in charts) and prototype implementation (tested with an example SparkPi application) of an external Task-aware scheduler. The answer to this RQ allows further experimental evaluation of the system (RQ3) with this new scheduler.

RQ3 - What is the impact on the performance of moving the scheduler outside of the Spark framework in the Kubernetes cluster?

To answer this RQ, we used external benchmarking tools to evaluate Spark runtime with a novel scheduler on extensive data executing TPC-DS queries. We compared and analyzed the runtimes of the original Spark and the modified version and concluded that moving the scheduler outside of the Spark framework in the Kubernetes cluster negatively impacts performance. The performance of the Apache Spark framework decreased significantly.

This RQ is answered by providing the results and their analysis of the experimental evaluation of the modified Spark version with the implemented prototype of the Task-aware scheduler.

6.2 Limitations and Threats to Validity

This section addresses some limitations and threats to the validity of this project.

Limitations

Important limitations of the work include scalability, generality, resource overheads and deployment complexity.

Scalability: We conducted this study on a limited scale, using specific cluster configurations and workloads. The scalability of the proposed solution on larger clusters and different workloads remains untested.

Generality: We tested the scheduler prototype with the SparkPi and TPC-DS queries, which may not cover the full range of use cases and real-world scenarios. Different types of workloads might demonstrate different performance characteristics.

Resource overheads: Implementing the external scheduler introduces additional network overheads, which are not considered in this study.

Development complexity: We did not evaluate the proposed scheduler's implementation complexity and maintenance overheads. The ease of use and integration into existing workflows may lead to potential challenges.

Threats to Validity

Threats to validity are described in terms of internal, external, and construct validity. We also assign potential reproducibility issues to the threat to validity.

Internal Validity: We optimized the configurations and setups for experiments to the best of our ability, but unaccounted variables might still affect the results.

External Validity: The findings from this study are based on a specific Kubernetes and Spark setup. Different versions or configurations might yield different results.

Construct Validity: The benchmarks and metrics used to evaluate performance might not capture all system efficiency and effectiveness aspects. Alternative metrics might provide additional insights.

Reproducibility: The experimental setup and environment specifics might pose challenges for the exact replication of the study. We tried to document the setup thoroughly, but minor differences can still lead to variations in results.

6.3 Directions for Future Work

The findings of this thesis can be extended to future works in several ways.

One direction for future work involves improvements in design and implementation. In this project, we implemented a scheduler that follows a limited FIFO approach in scheduling individual tasks on executors. This scheduler can be further modified to work on task Sets or even the whole DAG, applying more complex scheduling strategies and algorithms.

Another direction for future investment is improving the experimental evaluation. A more comprehensive range of benchmarks, real-world workloads, and larger clusters with varying configurations can be explored to evaluate the scheduler's performance better and provide additional insights into its limitations.

Incorporating this scheduler into other application frameworks like Dask or Knative is also essential. While doing so might involve additional challenges and design and core logic modifications based on specific frameworks, the general workflow should still apply to all runtime application frameworks.

Addressing these areas allows future research to build on this thesis's foundation and further advance the field of Kubernetes-based scheduling for Apache Spark and other applications.

References

[1] Statista. **Global Data Creation 2010-2025**, 2024. 1

[2] José María Cavanillas, Edward Curry, and Wolfgang Wahlster, editors. *New Horizons for a Data-Driven Economy - A Roadmap for Usage and Exploitation of Big Data in Europe*. Springer, 2016. 1

[3] Sameer Rajan and Apurva Jairath. **Cloud Computing: The Fifth Generation of Computing**. In *2011 International Conference on Communication Systems and Network Technologies*. IEEE, jun 2011. 1

[4] Peter M. Mell and Timothy Grance. **SP 800-145. The NIST Definition of Cloud Computing**. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, USA, 2011. 1

[5] Faraz Fatemi Moghaddam, Mohammad Ahmadi, Samira Sarvari, Mohammad Eslami, and Ali Golkar. **Cloud computing challenges and opportunities: A survey**. In *2015 1st International Conference on Telematics and Future Generation Networks (TAFGEN)*. IEEE, may 2015. 1

[6] Apache Software Foundation. **Apache Spark**. `https://spark.apache.org/`, 2018. 1, 2, 33

[7] Dask Developers. **Dask**. `https://www.dask.org/`, 2022. 1

[8] Knative Project. **Knative Documentation**, 2024. 1

[9] Kubernetes Authors. **Kubernetes**. `https://kubernetes.io/`, 2024. 1, 2, 34

[10] Carmen Carrión. **Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges**. *ACM Comput. Surv.*, **55**(7):138:1–138:37, 2023. 1

[11] Changpeng Zhu, Bo Han, and Yinliang Zhao. **A comparative performance study of spark on kubernetes**. *J. Supercomput.*, **78**(11):13298–13322, 2022. 1, 3

[12] Shwet Ketu, Pramod Kumar Mishra, and Sonali Agarwal. **Performance Analysis of Distributed Computing Frameworks for Big Data Analytics: Hadoop Vs Spark**. *Computación y Sistemas*, **24**(2), 2020. 2

[13] J. Dobies and J. Wood. *Kubernetes Operators: Automating the Container Orchestration Platform*. O'Reilly Media, 2020. 3, 19, 20

[14] Sacheendra Talluri, Nikolas Herbst, Cristina L. Abad, Tiziano De Matteis, and Alexandru Iosup. **ExDe: Design space exploration of scheduler architectures and mechanisms for serverless data-processing**. *Future Gener. Comput. Syst.*, **153**:84–96, 2024. 3

[15] Kubernetes Authors. **Operators**. `https://kubernetes.io/docs/concepts/extend-kubernetes/operator/`, 2023. 4

[16] Operator Framework. **Operator framework**. https://operatorframework.io/, 2024. 4, 20

[17] Zalando SE Sergey Vasilyev. **kopf**. https://kopf.readthedocs.io/en/stable/, 2024. 4, 11, 20

[18] Metacontroller. **Metacontroller**. https://metacontroller.github.io/metacontroller/intro.html, 2024. 4, 13, 20

[19] Datastrophic. **Core Concepts, Architecture, and Internals of Apache Spark**, May 22 2024. 4, 38

[20] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J. Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter A. Boncz. **LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms**. *Proc. VLDB Endow.*, **9**(13):1317–1328, 2016. 6

[21] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The AtLarge Vision on the Design of Distributed**

Systems and Ecosystems. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*, pages 1765–1776. IEEE, 2019. 7

[22] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan S. Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**. *CoRR*, **abs/2206.03259**, 2022. 8

[23] Nolar. **Kopf: Python Kubernetes Operator Framework**. `https://github.com/nolar/kopf`. 11

[24] Zalando SE. **Zalando Open Source: Scaling Open Culture**, 2024. 11

[25] TIOBE Software. **TIOBE Index for May 2024**, 2024. 11

[26] Teerapong Tanadechopon and Boontariga Kasemsontitum. **Performance Evaluation of Programming Languages as API Services for Cloud Environments: A Comparative Study of PHP, Python, Node.js and Golang**. In *2023 7th International Conference on Information Technology (InCIT)*, pages 293–297, 2023. 12, 21

[27] David Lion, Adrian Chiu, Michael Stumm, and Ding Yuan. **Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster?** In Jiri Schindler and Noa Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 835–852. USENIX Association, 2022. 12

[28] Zalando SE Sergey Vasilyev. **Alternatives**. https://kopf.readthedocs.io/en/stable/alternatives/, 2024. 13, 14, 22

[29] Google Cloud. **Google Kubernetes Engine (GKE): The most scalable and fully automated Kubernetes service**, 2024. 13

[30] Metacontroller Contributors. **Metacontroller**. `https://github.com/metacontroller/metacontroller/blob/master/README.md`, 2024. 13

[31] IBM. **What is Function as a Service (FaaS)?**, 2024. 13

[32] Red Hat. **What is a Webhook?**, February 1 2024. 13

[33] JSON.org. **JSON: JavaScript Object Notation**, 2024. 13

[34] Operator Framework Contributors. **Operator SDK**, 2024. 14, 15

[35] Go. **The Go Programming Language**, 2024. developed by a team at Google and many contributors from the open source community. 14, 21

[36] Mina Andrawos and Martin Helmich. *Cloud native programming with golang*. Packt Publishing, Birmingham, England, April 2023. 14

[37] Chandrakant. **What Are the Advantages and Disadvantages of Golang?**, 2024. 15

[38] Kubernetes SIGs. **Controller Runtime**, 2024. 15

[39] Alex Handy. **Build Your Kubernetes Operator With the Right Tool**. Red Hat Blog, January 13 2021. This is a guest post by Rafał Leszko, Integration Team Lead at Hazelcast. It contains the summary of the OperatorCon talk and the related blog post published at Hazelcast Blog. 15, 20, 22

[40] OperatorHub.io. **OperatorHub.io**, 2024. 15

[41] Luna Xu, Ali Raza Butt, Seung-Hwan Lim, and Ramakrishnan Kannan. **A Heterogeneity-Aware Task Scheduler for Spark**. In *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018*, pages 245–256. IEEE Computer Society, 2018. 16

[42] Muhammed Tawfiqul Islam, Satish Narayana Srirama, Shanika Karunasekera, and Rajkumar Buyya. **Cost-efficient dynamic scheduling of big data applications in apache spark on cloud**. *J. Syst. Softw.*, **162**, 2020. 16

[43] Dazhao Cheng, Xiaobo Zhou, Yu Wang, and Changjun Jiang. **Adaptive Scheduling Parallel Jobs with Dynamic Batching in Spark Streaming**. *IEEE Trans. Parallel Distributed Syst.*, **29**(12):2672–2685, 2018. 16

[44] Volcano Contributors. **Volcano Documentation**, 2024. 17

[45] The Apache Software Foundation. **Apache YuniKorn**, 2024. 17

[46] Cloud Native Computing Foundation. **Cloud Native Computing Foundation (CNCF)**, 2024. 17

[47] Yarn Contributors. **Yarn Package Manager**, 2024. 17

[48] Cloudera. **YuniKorn: A Universal Resources Scheduler**, 2024. 17

[49] Kubernetes Contributors. **kube-batch**, 2024. 17

[50] Shilin Wen, Rui Han, Ke Qiu, Xiaoxin Ma, Zeqing Li, Hongjie Deng, and Chi Harold Liu. **K8sSim: A Simulation Tool for Kubernetes Schedulers and Its Applications in Scheduling Algorithm Optimization**. *Micromachines*, **14**(3):651, 2023. 17

[51] Zeineb Rejiba and Javad Chamanara. **Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches**. *ACM Comput. Surv.*, **55**(7):151:1–151:37, 2023. 17

[52] Benjamin Schmeling and Maximilian Dargatz. *The Impact of Kubernetes on Development*, pages 1–57. Apress, Berkeley, CA, 2022. 19, 24

[53] Python Software Foundation. **Python**, 2024. 21

[54] The PHP Group. **PHP: Hypertext Preprocessor**, 2024. 21

[55] YAML.org. **YAML**, 2024. 24

[56] Muhammed Tawfiqul Islam, Satish Narayana Srirama, Shanika Karunasekera, and Rajkumar Buyya. **Cost-efficient dynamic scheduling of big data applications in apache spark on cloud**. *J. Syst. Softw.*, **162**, 2020. 33

[57] Holden Karau and Rachel Warren. *High Performance Spark*. O'Reilly Media, Sebastopol, CA, June 2017. 33

[58] The Apache Software Foundation. **Apache Spark**, 2024. 33

[59] Ojas Badwaik. **Component of Spark Application/Job**. `https://medium.com/@badwaik.ojas/component-of-spark-application-job-1a52b3ea2bc`, March 8 2023. 34

[60] The Apache Software Foundation. **Running Spark on Kubernetes**, 2024. 34, 35

[61] IBM. **Spark Application States**, January 19 2024. 34

[62] Kubernetes Authors. **Kubernetes**. https://kubernetes.io/docs/concepts/workloads/pods/, 2024. 34

[63] Learn.org. **What is Design Methodology?**, 2024. 36

[64] QuestionPro. **What is Applied Research?**, 2024. 36

[65] The Apache Software Foundation. **Apache Hadoop Fair Scheduler**, January 2024. 37

[66] H.B. Liu. **Spark Code Reading: Infrastructure and Pool**, May 2024. 38

[67] H.B. Liu. **Spark Code Reading: Task Scheduling Workflow**, May 2024. 38

[68] Fabric8. **Fabric8: Open Source Kubernetes Platform**, 2024. 39

[69] Matthijs Jansen, Linus Wagner, Animesh Trivedi, and Alexandru Iosup. **Continuum: Automate Infrastructure Deployment and Benchmarking in the Compute Continuum**. In Marco Vieira, Valeria Cardellini, Antinisca Di Marco, and Petr Tuma, editors, *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering, ICPE 2023, Coimbra, Portugal, April 15-19, 2023*, pages 181–188. ACM, 2023. 45

[70] Lennart Schulz. **ShareBench**, 2024. 46, 65

[71] Transaction Processing Performance Council. **TPC-DS Benchmark**, 2024. 46

[72] Kubernetes Documentation. **Minikube start**, 2024. 65

[73] Kopf Contributors. **Kopf Installation Guide**. `https://kopf.readthedocs.io/en/stable/install/`, 2024. 66

[74] Metacontroller Contributors. **Metacontroller Installation Guide**. `https://metacontroller.github.io/metacontroller/guide/install.html`, 2024. 66

[75] Operator Framework Contributors. **Operator SDK Installation Documentation**. `https://sdk.operatorframework.io/docs/installation/`, 2024. 66

Appendix A

Reproducibility

A.1 Abstract

Appendix A provides an artifact checklist, a description of hardware and software specifications, installation processes, and experiment workflow, and specifies the expected results when the experiment runs successfully.

A.2 Artifact Checklist (Meta-Information)

- **Algorithm: FIFO**
- **Program: Apache Spark, Kubernetes**
- **Run-time environment: Local machine, cluster**
- **Hardware local: Intel Core i7-8550U, RAM 16 GB**
- **Hardware cluster: Intel Xeon Silver 4210 CPU @ 2.20GHz, RAM 32 GB**
- **Metrics: Runtime, mean, median, IQR**
- **How much disk space required (approximately)?: 30g**
- **How much time is needed to prepare workflow (approximately)?: 15 minutes**
- **How much time is needed to complete experiments (approximately)?: 30 minutes**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: Apache License 2.0**

A.3 Description

This section describes how to access the code for the experiments and hardware and software specifications for the local environment and cluster.

How to Access

The modified Spark and Scheduler code for the experiments is available at GitHub repository.

Installation guides for the frameworks and link to Apache Source code:

- **Minikube** installation guide
- **Kopf** installation guide
- **Metacontroller** installation guide
- **Operator SDK** installation guide
- **Apache Spark v3.5** source code

Local Environmental Setup

The local environment setup, including hardware and software specifications, is shown in Table A.1.

Table A.1: Local system specifications

Component	Specification
Processor	Intel Core i7-8550U
CPU	Base clock speed: 1.80GHz
	Maximum turbo frequency: 2.00 GHz
RAM	16 GB
System Type	64-bit operating system, x64-based processor
Operating System	Windows 10 Home
Virtual Machine	Oracle VM VirtualBox 7.0.14
Kubernetes	Minikube 1.32.0
WSL	Ubuntu on Windows 20.04.4
Minikube VM Resources	4 CPUs, Memory: 8192 MB, Disk: 30720 MB
Apache Spark	v3.5.1

Cluster Setup

The cluster setup with its hardware and software specification is presented in Table A.2.

Table A.2: Cluster specifications

Component	**Specification**
Processor	Intel(R) Xeon(R) Silver 4210
CPU	Base clock speed: 2.20GHz
RAM	32 GB
Arhitecture	x86_64
Operating System	Linux
Virtual Machine	QEMU 6.1.0
Kubernetes	Minikube v1.27.0
WSL	Ubuntu on Windows 20.04.4
Apache Spark	v3.5.1

Modifications to Sharebench Framework

To run Sharebench [70] with a custom Spark version, several things have to be changed in the Sharebench source code. The code and commands are described in GitHub repository. The procedure for running the Sharebench experiment with custom Spark is the following:

1. Create custom Spark image based on apache/spark:3.5.1-scala2.12-java17-ubuntu image
2. Modify Dockerfile in the docker folder to build an image based on a custom Spark image
3. Apply the RBAC, cluster role binding configurations and CRDs
4. Run the benchmark

A.4 Installation

This section describes the installation of the Operator Frameworks and their prerequisites. Installation links were provided earlier in Appendix A.

Operator Frameworks Installation

Detailed installation guide of Kubernetes Development Environment - Minikube for various operating systems is available at the Kubernetes official website [72].

Installation guides of Kopf, Metacontroller and Operator SDK (does not support all Windows versions, but can be installed on a WSL from GitHub release) are available in the official documentation for Kopf [73], Metacontroller [74], and Operator SDK [75]. Prerequisites for each of the frameworks are listed below:

Table A.3: Frameworks prerequisites

Component	Specification
Kopf	pip installed, Python >= 3.8
Metacontroller	Kubernetes >= 1.17, kubectl available and configured
Operator SDK	curl installed, gpg >= 2.0

Go on WSL can be installed via the Go version manager (GVM v1.0.22). To install go1.20 - the latest version configured with Ubuntu on Windows v20.04.4, the go1.4 binaries must be installed, and then the version should be updated to a more recent one. The following steps were executed:

```
gvm install go1.4 -B
gvm use go1.4
export GOROOT\_BOOTSTRAP=$GOROOT
gvm install go1.20 -B
gvm use go1.20 --default
```

A.5 Experiment Workflow for Operator Frameworks Comparison

In this section, we describe the workflow of three frameworks - Kopf, Metacontroller and Operator SDK.

Kopf Workflow

The core logic is written in the file named *kopf.py*. Running it with *kopf run filename.py –verbose* allows monitoring a detailed description of the processes happening within the file.

```
kopf run kopf.py --verbose      # run code
kubectl apply -f crd.yaml       # create CRD
kubectl apply -f obj.yaml       # create CR object
kubectl get pods                # check the pods
```

```
    kubectl delete -f obj.yaml       # clean up - delete CR object
    kubectl delete pods --all        # clean up - delete all pods
```

To measure overhead, the Python code is changed to only print *CREATE POD IS RUNNING* instead of creating a new Pod. The procedure is repeated with the modified code.

Metacontroller Workflow

The procedure for Metacontroller is almost identical to Kopf, except that it also requires the creation of a controller. Since this framework is implemented with both Python and Go, the commands to run the code vary depending on the language. The core logic files are called *sync.py* and *sync.go*. When running *sync.go*, the path to the Kubernetes configuration file must be specified to avoid errors. Additionally, the command: ["python", "sync.py"] should be replaced with the command: ["./sync"] in the *controller.yaml* file. To run *sync.go*:

```
go run "your-path\sync.go" -kubeconfig="$env:USERPROFILE\.kube\config"
```

The procedure:

```
    python sync.py (or go)           # run code
    kubectl apply -f crd.yaml        # create CRD
    kubectl apply -f controller.yaml # create a controller
    kubectl apply -f obj.yaml        # create CR object
    kubectl get pods                 # check the pods
    kubectl delete -f obj.yaml       # clean up - delete CR object
    kubectl delete pods --all        # clean up - delete all pods
```

Similarly to Kopf, to measure an overhead, both codes are changed to print *CREATE POD IS RUNNING* instead of creating a new Pod. The procedure is then repeated with a modified code.

Operator SDK Workflow

Steps for creating and running an operator:

- Create directory: mkdir pod_creator_operator
- Go to that dir: cd pod_creator_operator

- Initialize new operator project: operator-sdk init –domain=your.domain –repo=github.com /yourusername
/pod-creator-operator
- Create a new API for Pod resource: operator-sdk create api –group=example –version=v1 –kind=Pod
- Modify file /pod_creator_operator/internal/controller/pod_controller.go
- Build and deploy an operator:
  - make generate
  - make manifests
  - make install
  - make docker-build docker-push (It might be needed to login to a docker account with docker login, then run docker tag example.com/pod-creator-operator:0.0.1 username/pod-creator-operator:0.0.1and docker push username/pod-creator-operator:0.0.1)
  - make deploy
- Create a new CR object file - sdk_pod.yaml
- Run kubectl apply -f sdk_pod.yaml
- To see the pods, run kubectl get pods
- To clean, run kubectl delete -f sdk_pod.yaml

Due to the complexity of modifying controller code, two operators with identical parameters were created instead of a single operator whose core logic code is manipulated to output a print statement rather than make a Pod. The only differences between the two Operators are the names of files and core logic (Pod creation versus print statement).

Apache Spark Procedure

The steps to run a modified Spark version with the SparkPi example are listed below. Note that the version of Kubernetes is downgraded. All files and commands are provided in the GitHub repository GitHub repository.

1. start Kubernetes cluster with v1.25.3

2. set custom Spark to be SPARK_HOME variable and include it in the PATH
3. build docker image
4. load an image to Minikube (or other Kubernetes cluster)
5. create Spark service account
6. apply RBAC configurations
7. apply cluster role binding
8. apply CRD
9. run an application starting the command with $SPARK_HOME/bin/spark-submit
10. in another terminal run the Scheduler
11. check log statements once the application terminates

A.6 Evaluation and Expected Results

For the Operators frameworks: when the experiment runs, it outputs the iteration number, like "Running iteration 54". If the experiment is terminated successfully, a CSV file is created with entries for 100 runtimes.

For the Spark with Task-aware scheduler: when running successfully, the log statements of the Spark source code output "Custom resources created for tasks", the Operator outputs the "Reached operator" statement and prints the task data. When the application stops running, it executes with exit code 0.

Appendix B

Self Reflection

Before working on the thesis, I had no prior knowledge of Apache Spark and Kubernetes. I had to learn many new concepts, such as containerization, docker, Kubernetes, Apache Spark, and even a new programming language - Go. This section reviews the challenges and things I have learned during this project.

At the beginning of my work, I was confused and overwhelmed by the amount of information I read about Kubernetes and Spark. Everything was abstract and used complex terminology I was not aware of. However, once I started implementing some of the concepts on my own, they slowly became more apparent. When I reread some of the materials that seemed very complex to me at the beginning, they suddenly became very logical and straightforward. I liked how this confusion insensibly transferred to an understanding of the topic.

When I started working with Spark source code, it took me additional time to explore how to build and create a source code distribution as I had never worked with open-source projects before and did not know the workflow. However, the most challenging part with Spark was getting errors that were not answered online and were not very descriptive. I had to consider which changes I made to my source code or environment and why this could potentially raise an error. Sometimes, I did not change anything, but the behaviour of the spark-submit command differed from the one I had on the previous day. I had to divide and conquer the errors, and I believe this approach has improved my skills as a Computer Science student.

I spent a couple of days solving one particular error. The error was due to incompatibility between the versions of my Spark (v3.5.1), Kubernetes client (v6.7.2), and Kubernetes itself, which was set up to the latest version but had to be downgraded to version 1.25.3.

There was barely any information about this online. So, I had to dive deep into the documentation of Spark and Kubernetes to find the solution.

I used several programming languages in this project: Python, Go, and Scala. While familiar with Python and Scala, I had to learn Go from scratch. I also learned YAML - a data serialization language that I later used outside the thesis project for a different course.

An important aspect that constantly added new challenges to my work was my local machine. My laptop operates on Windows, but many aspects of my work only work with Linux (like Operator SDK). Installation processes are more complex on Windows and require additional debugging. For example, Minikube installation on Windows gave me an error like "*This computer doesn't have VT-X/AMD-v enabled. Enabling it in the BIOS is mandatory*". It was challenging to solve as my BIOS settings did not have the options for "Advanced settings" or "Systems settings", where virtualization can be enabled, and there were no guides or manuals available for the BIOS of Xiaomi laptops. I solved this problem, but fixing it took me several days.

Additionally, starting the Minikube VM takes a lot of time on Windows. When I began working with Operator SDK, I switched to Ubuntu for Windows, which was way faster and did not have installation problems. However, it had its own flaws, as I had to work only with CLI, which was not always convenient.

Another problem with my local device was its computational power. It only has 16 GB of RAM, which was 100% in use for most of the time when I worked with Spark source code or ran Spark applications. As RAM was fully utilized but the memory was still needed, my laptop used disk memory, which slowed down the performance dramatically. Creating a distribution of the Spark version could take from 40 minutes to 2 hours.

One of the most important lessons I learned while working on this project was not only how to solve errors and bugs but also how to find information about them when no information is available. It made me think carefully about what exactly the program does and what might cause the problem, understanding the implementation more deeply.

I learned how to work with Kubernetes, which I will surely use in the future. I also got hands-on experience with a large open-source project that taught me how to work and read projects with extensive structure and how to make such a large project consistent with clear documentation, structure, and unified code readability.

For my future projects, it would be beneficial to document the working solution and the steps that have been tried out, even if there are many of them. I have had situations where the same error appeared, but the solution that worked previously did not work anymore, and I had to redo all the steps and find again all resources that have a potential solution. Having documented all steps initially would speed up this process significantly.

The process of working on the thesis project was well-structured and organized. I had sufficient communication and support from my supervisor and other atLarge team members, and I always knew the next steps I needed to take. I am grateful for the opportunity to work with this research team for the last four months.

In general, I enjoyed working on this project. It was interesting, challenging, but manageable. I gained a lot of essential knowledge about widely known frameworks and tools, structuring the work, dealing with a lot of information, debugging, and understanding big projects. I am sure I will use this knowledge and skills further in my career.