Implentation of an simplistic Interface for Big Data Workload Scheduler in Kubernetes

Implentation of an simplistic Interface for Big Data Workload Scheduler in Kubernetes

Lukas Schwerdtfeger

A thesis submitted to the
Faculty of Electrical Engineering and Computer Science
of the
Technical University of Berlin
in partial fulfillment of the requirements for the degree
Bachelor Technische Informatik

Berlin, Germany
December 22, 2015

Main supervisor:

Prof. Dr. habil. Odej Kao, Technical University of Berlin

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbst-
ständig und eigenhändig sowie ohne unerlaubte fremde Hilfe
und ausschließlich unter Verwendung der aufgeführten
Quellen und Hilfsmittel angefertigt habe.

Berlin, den

Zusammenfassung

Kurze Zusammenfassung der Arbeit in 250 Wörtern.

Abstract

Short version of the thesis in 250 words.

Acknowledgements

This chapter is optional. First of all, I would like to...

Contents

List of Figures

List of Tables

1

Introduction

1.1 Motivation

The current population is producing more and more data. This creates an excellent opportunity for many businesses. Businesses willing to profit from collected data by using it to improve their sales strategies have to collect and store GigaBytes and upwards to ExaBytes of data. With storage costs becoming more affordable, companies are even less likely to toss away potential valuable data, creating so-called Data Lakes.

Collecting data is only the first step. It takes many stages of processing, through aggregation and filtering, to extract any meaningful information. Usually, the sheer mass of collected data makes it not very useful, to begin with.

Unfortunately, when working with ExaBytes of data, it is no longer feasible to work on a single machine. Especially when dealing with a stream of data produced by a production system and the information collected from yesterday's data is required the next day, or ideally immediately.

Scaling a single machine's resources to meet the demand is also not feasible. It is either very expansive or might just straight up not be possible. On the other hand, cheap commodity hardware allows the scaling of resources across multiple machines is much cheaper than investing in high-end hardware or even supercomputers.

The complexity of dealing with a distributed system can be reduced using the abstraction of a cluster. A **Cluster Resource Manager** is used, where a system of multiple machines forms a single coherent cluster that can be given tasks to.

Stream Processing of Data across such a cluster can carry out using Stream or Batch Processing frameworks, such as Apache Spark or Apache Flink. These frameworks already implement the quirks of dealing with distributed systems and thus hide the complexity.

The problem is that multiple Batch Jobs running on a single cluster need resources that need to be allocated across the cluster. While Resource Allocation is the Task of the Cluster Resource

Manager, the manager usually does not know how to allocate its resources optimal and often requires user to specify the resources that should be allocated per job. This usually leads to either too little resources being allocated per job, starving jobs and increasing the runtime, or more often over-committing resources and thus leaving resources in the cluster unused.

Another problem that arises is the fact that even though the reoccurring nature of Batch Jobs, not all Batch Jobs use the same amount of Resources. Some are more computationally intensive and require more time on the CPU, while others are more memory intensive and require more or faster access to the machine's memory. Others are heavy on the I/O usage and use most of the system's disk or network devices. This shows up in vastly different Job runtime (also total runtime) depending on the Scheduling of Batch Jobs across the Cluster.

Finding an intelligent Scheduling Algorithm that can identify reoccurring Jobs and estimate their resource usage based on collected Metrics and thus create optimal scheduling is not an easy task. It also requires a lot of setup when dealing with a Cluster Resource Manager.

1.2 Problem Description

Cluster Resource Managers, like YARN, emerging from Apache Hadoop, were centered around Batch frameworks.

With the rise of Cloud Computing and all the benefits that come with it, companies were quick to adopt new cloud computing concepts. The Concept of a Cluster Resource Manager introduced a notion of simplicity to those developing applications for the cloud. A Cluster Resource Manager now managed many aspects that used to be handled by dedicated Operations-Teams.

Kubernetes, a Cluster Resource Manager that was initially developed by Google, after years of internal use, provided an all-around approach to Cluster Resource Management for not just Batch-Application. The global adoption of Kubernetes by many leading companies, led to the growth of the ecosystem around it. Kubernetes has grown a lot since and has become the new industry standard, benefiting from a vast community.

Old Batch-Application-focused Cluster Resource Managers that used to be the industry standard are being pushed away by Kubernetes. Unfortunately, vastly different Interfaces or Scheduling Mechanism between other Cluster Resource Managers usually block the continuation of existing research done in the field of Batch Scheduling Algorithms.

Finding an efficient scheduling algorithm is a complex topic in itself. Usually, the setup required to further research existing scheduling algorithms is substantial. Dealing with different Cluster Resource Manager further complicates continuing on already existing work.

1.3 Goal of this Thesis

To aid further research in the topic of Batch-Scheduling-Algorithms, the goal of this thesis is to provide a simplistic interface for Batch-Scheduling on Kubernetes.

Already existing Scheduling Algorithms, like Mary and Hugo, were initially developed for the Cluster Resource Manager YARN. Reusing existing Scheduling Algorithms on the nowadays

broadly adopted Cluster Resource Manager Kubernetes is not a trivial task due to vastly different interfaces and interaction with the Cluster Manager.

Extending existing research to the more popular Resource Manager Kubernetes provides multiple benefits.

1. Research on Scheduling Algorithms for YARN has become less valuable due to less usage
2. The large ecosystem around Kubernetes allows for a better development environment due to debugging and diagnostic tooling
3. Initial setup of a Kubernetes cluster has become smaller due to applications like MiniKube, which allows a quick setup of a cluster in the local machine and Cloud Providers offering Kubernetes Clusters as a service.

The interface should provide easy access to the Kubernetes Cluster, allowing an External-Scheduler to place enqueued Batch-Jobs in predefined slots inside the cluster.

For an External-Scheduler to form a scheduling decision, the interface should provide an overview of the current cluster situation containing:

1. Information about empty or in use slots in the cluster
2. Information about Jobs in the Queue
3. Information about the history of reoccurring Jobs, like runtime

It should be possible for an External-Scheduler to form a scheduling decision based on a queue of jobs and metrics collected from the cluster. The interface should accept the scheduling decision and translate it into Kubernetes concepts to establish the desired scheduling in the cluster.

Currently, the Kubernetes Cluster Resource Manager does not offer the concept of a Queue. Submitting jobs to the cluster would either allocate resources immediately or produce an error due to missing resources.

Kubernetes does not offer the concept of dedicated Slots for Applications either. While there are various mechanisms to influence the placement of specific applications on specific Nodes, these might become unreliable on a busy cluster and require a deep understanding of Kubernetes concepts, thus creating a barrier for future research.

1.4 Structure of this Thesis

The structure of this thesis allows the reader to read it in any order. To guide the reader through this thesis, the structure of this Thesis section will briefly explain which section contains which information.

The Background Chapter is supposed to give a brief overview of this thesis's underlying concepts. This chapter introduces Big Data Streaming Processing, the Cluster Resource Manager Kubernetes, and Scheduling.

Following the Background chapter, the thesis provides an overview of the approach taken to tackle the problem described in the Problem Description Section. The Approach Section focuses on more profound concepts of Kubernetes and the Scheduling Cycle of the Kubernetes Scheduler. It summarizes the Kubernetes Operator pattern, which is commonly used to extend Kubernetes.

Implementation details will be given inside the Implementation Chapter, where an architectural overview and interaction between individual components are explained. The Implementation section also emphasizes the design Process for the Interface, which is exposed to an External-Scheduler. A significant part of the implementation is the Operator, which will be discussed extensively. The Implementation chapter shows how the points made inside the Approach Chapter are were implemented in the end.

An Evaluation of the research and contribution done by this thesis will be presented inside the evaluation chapter. Here its functionality is demonstrated. This section will also outline some of the limitations.

Before concluding the thesis, a comparison between State of the Art Technology is made.

2

Background

2.1 Distributed Dataflow Applications

Distributed Dataflow Processing aims to solve the problem of analyzing large quantities of data. In the last years, the amount of generated data has exploded. This creates a problem where single machines can no longer analyze the data in a meaningful time. While the Big Data Processing frameworks still work on single machines, computation is usually distributed across many processes running on hundreds of machines to analyze the data in an acceptable time.

Analyzing data on a single machine is usually limited by the resources available on a single machine. Unfortunately, increasing the resources of a single machine is either not feasible from a cost standpoint or simply impossible. There is only a limited amount of processor time, memory, and I/O available. Cheap commodity hardware allows a cluster to bypass the limitations of a single machine, scaling to a point where the cluster can keep up with the generated data and once again analyze data in a meaningful time frame.

Dealing with distributed systems is a complex topic in itself. Many assumptions that could be made in a single process context are no longer valid. Scaling to more machines increases the probability of failures. Distributed Systems need to be designed to be resilient against hardware-failures, network outages/partitions and are expected to recover from said failures. Having a single loss resulting in no or an invalid result will not scale to systems of hundreds of machines, where it is unlikely to not encounter a single failure during execution.

Scaling computational resources beyond the limitation of single machines is not a new problem. High-Performance-Computing (HPC), like supercomputers, are already commonly used for computational heavy use cases. Distributed Dataflow Applications initially used the static partitioning approach and evolved with the advance of cloud computing and cluster management systems to scale more dynamically, depending on the users' needs. Distributed Dataflow Applications may be classically categorized as HPC applications bringing high performance and efficiency due to sophisticated scheduling, where the cloud offers resiliency, elasticity, portability, and manageability.[**?**]

Distributed Dataflow frameworks can be put into two categories, although many fall in both categories: Batch Processing and Stream Processing. In Batch Processing, data size is usually known in advance, whereas Stream Processing expects new data to be streamed in from different sources during the runtime. Batch Processing jobs will complete their calculation eventually, and Stream Processing may run for an infinite time.

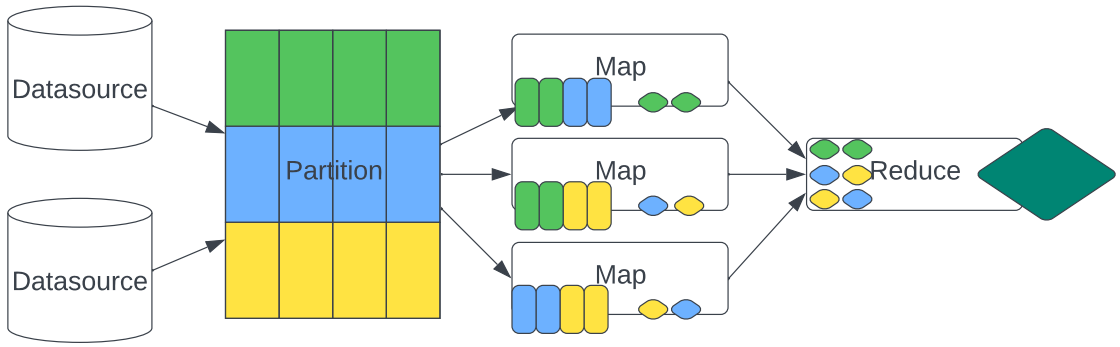


Figure 2.1: Example of Spark DAG [**?**]

Internally, Big Data Processing frameworks build a directed acyclic graph (DAG) of stages required for the analysis. Stages are critical for saving intermediate results, to resume after failure, and are usually steps during the analysis, where data moving across processes is required. A commonly used programming model is the MapReduce Programming model[**?**], where stages can be generalized in Map and Reduce operations.

Map operations can be performed on many machines in parallel, without further knowledge about the complete data sets, like extracting the username from an application-log-line. Reduce operations require moving data around the cluster. These are usually used to aggregate data, like grouping by a key, summing, or counting.

Datasets that the Distributed Processing frameworks analyze are usually in the range of terabytes which is multiple magnitudes higher than the amount of memory that each machine has available. partitioning of the data is required due to the limitations of each single machine, while persistent storage, like hard-drives, might be closer to the extent of BigData, computation will quickly become limited by the amount of I/O a single machine can perform. Commonly a Distributed Dataflow Application is combined with a distributed file system (like HDFS[**?**]) that spans a shared filesystem across many machine. In a distributed filesystem, the filesystem is usually broken down into smaller chunks, where every machine holds a few chunks of the complete filesystem as part of their local filesystem. If data from the distributed filesystem is not local to the machine' filesystem, data will be moved across the network. Replicating chunks increases fault tolerance and thus prevents the loss of data but also requires synchronization across replicated chunks once they get updated. Data-Locality describes how close the data is, which the current task requires. If a task has good data-locality on a machine, no data movement is required.

The user of Big Data Processing frameworks is usually not required to think about how an efficient partition of the data across many processes may be accomplished. Frameworks are

designed in a way where they can efficiently distribute a large amount of work across many machines.

2.2 Scheduling

In general, scheduling is the process of assigning resources to a task. This includes the question:

1. Should any resources be allocated for the task at all?
2. At which point in time should resource be allocated?
3. How many resources should be allocated?
4. Which of the available resources should be allocated?

Scheduling is essential for Operating Systems that need to decide which process should get CPU time and which processes may need to wait to continue computation. In the case of multiple CPUs, a decision has to be made on which CPU should carry out the computation. The Operating System is not just concerned with CPU-Resources, but also I/O Device resources. Some devices may not work under concurrent usage and require synchronization.

In some cases, a simple FIFO scheduling that works on tasks in order there were submitted produces acceptable results. Scheduling depends on a goal. Some algorithms aim to find the optimal schedule to respect any given deadlines. Whereas some distinguish between Soft and Hard Deadlines, where ideally no deadlines would be missed at all, occasionally missing soft deadlines to guarantee Hard deadlines are met is acceptable. In general, finding a single best schedule that allows resources to be allocated optimally is not possible. Scheduling for a fast response time or throughput might prefer shorter tasks to be run, when possible, and might starve longer running tasks for a long time before progress can be made.

Common goals of scheduling are [**?**]:

- ◇ Minimizing the time between submitting a task and finishing the task. A Task should not stay in the queue for a long time and start running soon after submission.
- ◇ Maximizing resource utilization. Resources available to the scheduler should be used, even if that means skipping tasks waiting in the queue.
- ◇ Maximizing throughput. Finishing as many jobs as possible may starve longer running jobs in favor of multiple shorter running jobs.

To achieve their goals scheduling algorithm might allow preemption, where the currently active task could be preempted for another task to become active. Some scheduling algorithms account for the potential overhead of preempting the current task (like a context switch).

Scheduling can be applied at multiple levels of the Software Stack, like scheduling processes at the Operating System level, Virtual Machines at the Hypervisor level, or scheduling tasks across many applications running on many machines at the cluster level. The higher up the stack, the more and more potential schedules become possible. It is a balancing act of complexity and performance. At the highest level, the scheduler has the most knowledge about the systems and could come up with optimal scheduling. However, it seems to be a wise choice for scheduling to be handled in their respective stack layer since a lot of complexity can be encapsulated inside each layer. Allowing the Top Level Scheduler to micromanage the complete stack creates an explosion of complexity.

For this work, the scope of scheduling is limited to Batch Scheduling of Jobs in a cluster.

The Question of Scheduling in a Distributed System is now the question of which machines resources should be used for which job. In Batch Scheduling algorithms need to pay attention to the characteristics of a Distributed System:

1. Potential heterogeneity of the system, with machines of different Hardware and different Operating Systems or Software
2. Spontaneously adding and removing resources of the Cluster
3. Interference between Applications residing on the same machine, same rack, same network switch, etc. (CO-Location)

While some of these factors can be controlled, different algorithms can be chosen for various use cases.

In Distributed Dataflow Applications, we deal with considerable different levels of scheduling. At the Framework level, applications build a DAG-based execution plan based on the job submitted. The initial DAG breaks done the job into their respective Map and Reduce Operations. These Operations will be broken down further into smaller Tasks based on the Partitioning of Data. Finally, Tasks are executed on an arbitrary number of processes across different machines. Optimizing the schedule of tasks to an executor process will be called DAG-Level scheduling and may now also include factors like Data-Locality.

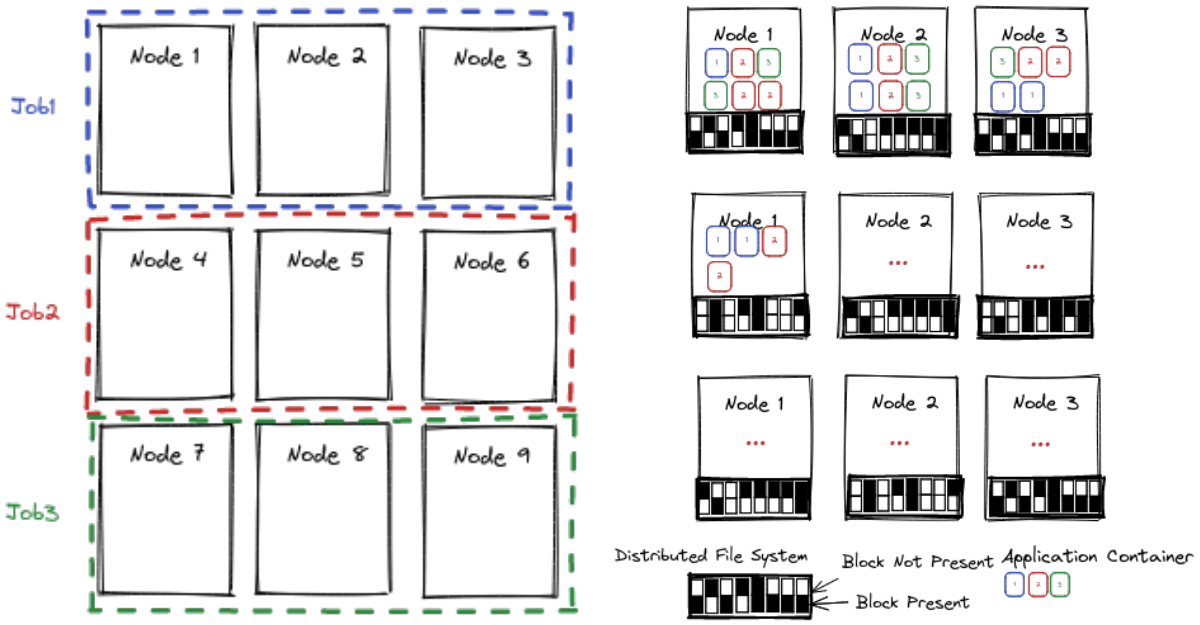
Moving Up one Level Higher in the stack, we are concerned with running multiple Jobs inside the same cluster, and a decision needs to be made which job can spawn their executor on which Nodes. Executors are packaged on Containers. The containers are isolated so that they can not access each other.

A more profound introduction into the specifics of scheduling in Kubernetes is inside the Scheduling in Kubernetes chapter.

Cluster Resource Manager like Mesos[**?**] allows scheduling on multiple levels, with Resource Offers. The top-level Mesos scheduler finds multiple candidates and offers them to the DAG-Level scheduler framework. The potential for cross-level scheduling would allow the Top-level scheduler to respect data-locality constraints for the DAG-level scheduler.

2.3 Cluster Management Systems

HPC supercomputers are commonly built with expensive hardware and tightly coupled operating systems, allowing parallel applications to take advantage of resources available on the supercomputer. supercomputers are carefully planned with software designed for specific purposes. This usually meant that multiple applications running on a supercomputer would have resources statically partitioned. Coarse grain partitioning introduces inefficient use of hardware once part of the statically partitioned resources is no longer in use.



With the emergence of Distributed Dataflow Applications, like Hadoop MapReduce[**?**], the need for a more fine-grain partitioning of cluster resources came along. The move away from supercomputers to cheap commodity hardware meant that clusters could be scaled up or down easily, where previously careful planning was required. Managing a dynamic system of potentially thousands of machines can not be done in a manual fashion. A cluster resource manager is required to build the abstraction of a single cohesive cluster that can be tasked with jobs.

Hadoop is one of many Open-Source MapReduce implementations. Cluster computing using commodity hardware was driven by the need to keep up with the explosion of data. The initial version of Hadoop was focused on running MapReduce jobs to process a web crawl [**?**]. Despite the initial focus, Hadoop was widely adopted evolved to a state where it was no longer used with its initial target in mind. Wide adoptions have shown some of the weaknesses in Hadoops architecture: - tight coupling between the MapReduce programming model and cluster management - centralized Handling of jobs will prevent Hadoop from scaling

The tight coupling leads Hadoop users with different applications to abuse the MapReduce programming model to benefit from cluster management and be left with a suboptimal solution. A typical pattern was to submit ‘map-only’ jobs that act as arbitrary software running on top of the resource manager. [**?**] The other solution was to create new frameworks. This caused the invention of many frameworks that aim to solve distributed computation on a cluster for many different kinds of applications [**?**]. Frameworks tend to be strongly tailored to simplify solving specific problems on a cluster of computers and thus speed up the exploration of data. It was expected for many more frameworks to be created, as none of them will offer an optimal solution for all applications.[**?**]

Initially, frameworks like MapReduce created and managed the cluster, which only allowed a single application across many machines—running only a single application across a cluster of machines led to the underutilization of the cluster’s resources. The next generation of Hadoop allowed it to build ad-hoc clusters, using Torque [**?**] and Maui, on a shared pool of hardware.

Hadoop on Demand (HoD) allowed users to submit their jobs to the cluster, estimating how many machines are required to fulfill the job. Torque would enqueue the job and reserve enough machines once they become available. Torque/Maui would then start the Hadoop master and its slaves, subsequently spawning the Task and JobTracker that make up a MapReduce application. Once all tasks are finished, the acquired machines are released back to the shared resource pool. This can create potentially wasteful scenarios, where only a single reduce task is left, but many machines are still reserved for the cluster. Usually, resources requirements are overestimated and thus leaves many clusters resources unused[**?**].

With the introduction of Hadoop 3, the MapReduce Framework was split into the dedicated resource manager YARN and MapReduce itself. MapReduce was no longer running across a cluster, but it was running on top of YARN, who manages the cluster beneath. This allows MapReduce to run multiple jobs across the same YARN cluster, but more importantly, it also allows other frameworks to run on top of YARN. This moved a lot of complexity away from MapReduce and allowed the framework to only specialize on the MapReduce programming model rather than managing a cluster. This finally allows different programming models for Machine Learning tasks that tend to perform worse on the MapReduce programming model [**?**] to run on top of the same cluster as other MapReduce jobs.

Using the resource manager YARN allows for a more fine-grain partitioning of the cluster resources. Previously a static partitioning of the clusters resources was done to ensure that a specific application could use a particular number of machines. Many applications can significantly benefit from being scaled out across the cluster rather than being limited to only a few machines. - Fault tolerance: frameworks use replication to be resilient in the case of machine failure. Having many of the replicas across a small number of machines defeats the purpose - Resource utilization: frameworks can scale dynamically and thus use resources that are not currently used by other applications. This allows applications to scale out rapidly once new machines become available - Data-locality: usually, the data to work on is shared across a cluster. Many applications are likely to work on the same set of data. Having applications sitting on only a few machines, but the data to be shared across the complete cluster leads to a lousy data locality. Many unnecessary I/O needs to be performed to move data across the cluster.

Fine-grain partitioning can be achieved using containerization, where previously applications were deployed in VMs, which would host a full operating system. Many containers can be deployed on the same VM (or physical machine) and share the same operating system kernel. The hosting operating system makes sure applications running inside containers are isolated by limiting their resources and access to the underlying operating system.

Before Hadoop 3 with YARN was published, an alternative cluster manager Mesos was publicized. Like YARN, Mesos allowed a more fine granular sharing of resources using containerization. The key difference between YARN and Mesos is how resources are scheduled to frameworks running inside the cluster. YARN offers a resource request mechanism, where applications can request the resources they want to use, and Mesos, on the other hand, offers frameworks resources that they can use. This allows frameworks to decide better which of the resources may be more beneficial. This enables Mesos to pass the resource-intensive task of online scheduling to the frameworks and improve its scalability.

The concept of containerization, which offers a lightweight alternative to traditional VMs, has

been internally used by Google many years before YARN and Mesos. Google has used container orchestration systems internally for multiple years before releasing the open-source project Kubernetes[**?**]. Kubernetes was quickly adopted and has become the defacto standard for managing a cluster of machines. Kubernetes offers a descriptive way of managing resources in a cluster where manifests describing the cluster's desired state are stored inside a distributed key-value store[**?**] etcd. Controllers are running inside the cluster to monitor these manifests and do the required actions to bring the cluster into the desired state. With Kubernetes offering building blocks and the mechanism of a control loop, the Operator pattern in combination with Custom Resource Definitions (CRDs) is commonly used to extend Kubernetes functionalities. Where Hadoop's YARN was focusing on Distributed Dataflow Applications, Kubernetes enables developers of all kinds of applications the scalability and resilience of the cloud.

3

Approach

The Approach section gives a more profound overview of core concepts of Kubernetes, which intern are the building blocks of the work done in this thesis. Starting with the Operator Pattern, which is strongly connected with the method, Kubernetes achieves its cluster orchestration, the *control loop*. The second section details how scheduling works in Kubernetes and how it can be manipulated to be controlled by an External-Scheduler using the Interface.

3.1 Extending Kubernetes using the Operator Pattern

A complete introduction into the core concepts of Kubernetes does not fit in the scope of this thesis. Commonly used concepts of Kubernetes, like *Deployments*, *ReplicaSets*, *Jobs*, and *DaemonSets* are already well documented[**?** **?**].

The smallest unit of deployable software in Kubernetes is a Pod[**?**]. While a Pod may consist of multiple containers, containers in a Pod are guaranteed to run on the same Node. Containers in Pods also share storage and network resources across the container boundary. In general, containers inside the same Pod are tightly coupled and commonly used in a sidecar pattern to extend the main container with common functionalities across the cluster, like the Kube-RBAC-Proxy[**?**] that is frequently used with containers that interact with the Kubernetes API and require authorization.

Usually, in Kubernetes, Pods are not created by themself but are managed by resources that build on top of them. Most commonly, Pods are used in combination with Jobs, Deployments, or StatefulSets, which control the lifecycle of the Pod.

The Operator pattern is based on the already existing design used by Kubernetes native resources, the *control loop*. Resources like Jobs, Deployments[**?**], and ReplicaSets[**?**] describe a desired state of Pods. Controllers take control over resources and manage them automatically. Controllers are not tied to the lifetime of a particular resource, but they exist and monitor the cluster state at all times in a control loop. Figure 3.1 demonstrates the control loop of the ReplicaSet controller. As the name implies, the ReplicaSet controller manages ReplicaSet

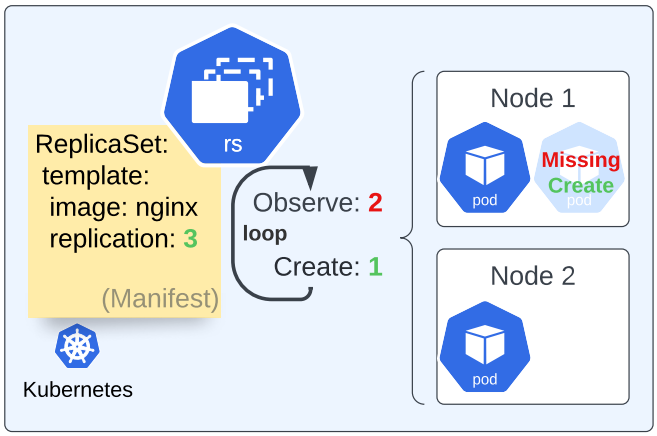


Figure 3.1: ReplicaSet control loop

resources, consisting of a pod template with the number of replicas. The ReplicatSet controller makes sure that the exact amount of Pods that correspond to the template are healthily running. If any of the Pods fails, the ReplicaSets controller deletes it and creates a new one. The controller also knows if the resources manifest is updated and updates the replication of the Pod accordingly.

Other Kubernetes resources like Deployments are built on top of ReplicaSets. The Deployment manages ReplicatSets to create the desired replication. If the Deployment is updated to a more recent version, all containers need to be replaced with the newer version. Restarting all Pods at once would leave the application in an inaccessible state. Deployments support many policies that dictate how these actions should happen. A common policy for Deployment is a rolling deployment, where old Pods coexist with new Pods until all new Pods are healthily running. The Deployment creates an additional ReplicaSet for the new version and simultaneously decrements the old ReplicaSet for each new Pod.

Using multiple controllers, of which each is only concerned with a single resource, creates a very loosely coupled and thus highly extendible interface. The idea of a reconciliation controller loop improves the resiliency of a system by observing the current state, comparing it to the desired state, and taking actions for the observed state to converge into the desired state.[**?**]

The Operator pattern is commonly used to extend Kubernetes functionalities. Most of the time, however, just creating a new controller is not enough to extend Kubernetes. It usually also requires Custom Resource Definitions (CRDs). A CRD is the Kubernetes way of defining new resources that are allowed to exist in the cluster. Having multiple controllers listing to the same resource, like a Deployment, makes little sense or could even cause issues. Thus the combination of a new CRD and a controller that knows how to handle it creates the *Operator pattern*. The term Operator is used as the controller is designed to replace previously manual work of configuring Kubernetes native resources done by an Operator. An everyday use case for the Operator pattern is to control applications at a higher level, where previously multiple deployments and services may have been required to operate a database. The Operator pattern could reduce that to just a single manifest containing the meaningful configuration. Operators

can thus be created by experts operating the software and be used by any Kubernetes cluster.

Common best practices when developing Operators are that Operators should only manage a single kind of resource, and multiple Operators should be used if multiple resources need to be managed. Rather than micro-managing resources down to the creation of Pods, Operators should delegate to existing Operators where ever possible, e.g., using Kubernetes native resources, like Deployments. Multiple Operators should not act on the same CRD, as this requires synchronization between Operators and enforces tight coupling between components. Since all Operators work with the Kubernetes API, a lot of identical boiler code has been written over the years, and many best practices are contained inside opinionated frameworks[**?**]. Operators implemented in the context of this thesis are using the Java Operator SDK.

3.2 Scheduling in Kubernetes

In this section, the scheduling model of Kubernetes will be introduced as it is a vital part of the implementation for the External-Scheduler-Interface. The scheduling problem in Kubernetes is the problem of deciding which Pods are running on which machine. Kubernetes runs workloads by placing containers into Pods to run on Nodes. Depending on the cluster, a Node may be a virtual or physical machine. Each Node is managed by the control plane and contains the necessary software to run Pods [**?**]. For some Pods, the question can be easily solved. For example, Pods controlled by a DaemonSet [**?**] are by the specification of the DaemonSet running on every Node. Without further information, a feasible choice of scheduling Pods onto Nodes seems to be simple round-robin scheduling. Every Pod requiring scheduling gets scheduled onto the next Nodes until all Nodes have Pods then the cycle is repeated. However, both Pods and Nodes can influence the scheduling.

Pods can specify the resources they will use and may even set a hard requirement in the form of a request for resources they require to run. Technically the Pod does not specify any resources, but any of its containers can. Kubernetes distinguishes between resource requests and resource limits. A limit is enforced by the container runtime and prevents a container from using more resources than its configuration allows. A resource request needs to be guaranteed by the scheduler but does not prevent the Pod from using more resources than requested. The resource request is a hard requirement for scheduling, whereas the limit can be disregarded[**?**].

Pods can also directly influence the Node they should be scheduled on, either through specifying a *NodeName* directly, a *NodeSelector*, which identifies possible Nodes, or a more general concept of *affinity*. Affinities provide the ability to set hard and soft requirements, where a Pod may become unschedulable if a hard requirement cannot be met, and a soft requirement is not preferred but an acceptable decision. With Affinities, even inter-pod Affinities can be specified and Pods can choose not to be scheduled on a Node if another Pod is already deployed.

On the other side, Nodes can also specify so-called Taints, where only Pods with the fitting Toleration can be scheduled onto the Node. Kubernetes also use the Taint mechanism to taint Nodes that are affected by network problems, memory-pressure, or are not ready after a restart. The Taint specifies if just new Pods cannot be scheduled *NoSchedule* onto the Node, or if even already running Pods should be evicted *NoExecute*. Tainted Nodes with either type of Taints are not chosen during scheduling.

However, Pods can bypass the Taint Node if they have the fitting Toleration. For example, if a Node becomes unreachable due to a network outage, Nodes affected will be tainted with the *node.kubernetes.io/unreachable"* Taint, which is set to *NoExecute*. Pods can neither be scheduled onto the unreachable Nodes nor keep running without the fitting Toleration. In a likely scenario where the network recovers and the machine becomes reachable again, it would be inefficient to restart all Pods instantly if a machine becomes unreachable for a short time. To prevent Pods from being evicted (and thus most likely restarted), Kubernetes by default creates the unreachable Toleration for all Pods. However, the Toleration set by Kubernetes also specifies a maximum duration where the taint is tolerated (by default, 5 minutes). If a Node stays unreachable for more than 5 minutes, all Pods are evicted.

In the context of scheduling, Taints, Tolerations, Pods, and Nodes only refer to the objects stored inside the distributed key-value store. If a physical machine becomes unreachable, the objects manifest is updated. Whether the physical containers are still running or not cannot be determined, but for the duration of the Toleration, they do count as running and are not replaced with new Pods.

3.2.1 Scheduling Cycle

Kubernetes scheduling is based on the Kubernetes Scheduling Framework. As the extensible nature of Kubernetes, the scheduling framework is also extensible. Customization to the scheduling can be done at so-called extension points.

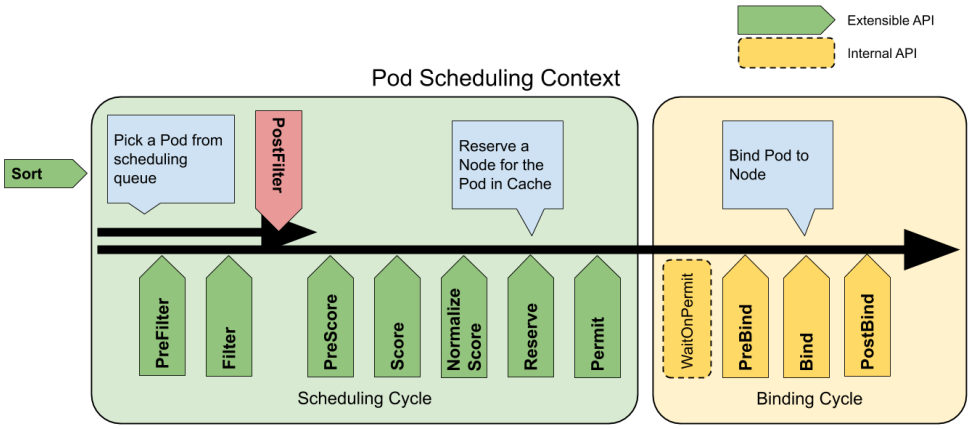


Figure 3.2: Scheduling Cycle

Each attempt of scheduling Pods to Nodes is split into two phases, the *scheduling cycle*, and the *binding cycle*.

The scheduling cycle is more interesting in the context of this work. In this part of the cycle, the decision is made on which Node to schedule which Pod. The Binding cycle is the scheduling phase, where the Pod is being deployed to the physical machine. Multiple scheduling cycles will not run concurrently, as this would require synchronization. Since the scheduling cycle is relatively fast, executing multiple scheduling cycles in parallel seems unnecessary. However, the binding cycle requiring the deployment of Pods is compared to the scheduling cycle rather

long-living and thus may be executed in parallel. Both the scheduling cycle and the binding cycle may abort the scheduling of a Pod in case it may be unschedulable.

The Pod scheduling context can be modified through plugins that intern use the scheduling framework's extension points. The graphic 3.2 shows the available extension points, which are the Pod scheduling cycle phases. Multiple plugins can be registered for any of the different extension points and will run in order.

This section will give a brief breakdown of the different stages and highlight those critical in the implementation of the External-Scheduler-Interface.

Queue Sort: Sometimes, multiple Pods require a scheduling decision. The queue sort extension point extends the scheduling cycle with a comparison function that allows the queue of Pods to wait for a scheduling decision to be sorted. Usually, multiple plugins can influence the scheduling decision at a time; however, having numerous plugins sorting the queue of waiting Pods will not result in anything meaningful. Thus, only a single plugin can control the queue at a time. After sorting the queue of pods, the first Pod is chosen and passed onto the next stage.

Filter: The scheduling decision is made by first using the pre-filter and filter extension point. The filter extension point filters out Nodes that cannot run the Pod. Most extension points also have a pre-extension point used to prepare information about the Pod. As mentioned earlier, any extension point can return an error indicating that the Pod may be unschedulable.

The filter extension point also has a **Post-Filter** extension, which is only called if no Node passes the filter, and a Pod becomes *unschedulable*. The post-filter extension can then be used to find a possible scheduling using preemption. Preemption is very important, as it is required to model Pods having a higher priority than other Pods. In this work, preemption is required to guarantee that the scheduling from an External-Scheduler is correctly applied even if other Pods in the cluster exist that are unknown to the External-Scheduler.

Scoring: The filter plugin is concerned with hard requirements that prevent Pods from being scheduled. If previous filter plugins deem multiple Nodes suited for the Pod to be scheduled on, a decision needs to be made to find the best Node. The scoring plugin considers soft requirements and gives Nodes a lower or higher score that can or cannot fulfill the soft requirements. Finally, all scores are normalized between two fixed values.

Reserve: The reserve phase is used to reserve resources on a Node for a given Pod. This is required due to the asynchronous transition into the binding cycle. Pods that are supposed to be bound to a Node invoke reservation to prevent possible race conditions between future scheduling cycles. Reservation may fail, in which case the cycle moves to the unreserve phase, and all previously completed reservations are revoked. Usually, the reservation extension point is used for applications that will not use the default containerization mechanism of Kubernetes and rely on different binding mechanisms.

Permit: The final stage of the scheduling cycle is the permit stage, which ultimately denies or delays the binding of a Pod in a case where binding might not be possible or still requires time.

3.2.2 Extending the Scheduler

Common ways of extending the scheduler are either to implement a custom scheduler and to replace the KubeScheduler in the cluster, or to use the *Extender* API that instructs the scheduler to invoke an external API for its scheduling decision.

The pluggable architecture of the scheduler allows plugins to extend the scheduling cycle at the extension points. Different scheduler profiles can use different plugins. Profiles can be created or modified using a *KubeSchedulerConfiguration*. Only Pods that specify the scheduler profile using *.spec.schedulerName* are scheduled using the profiles.

The Extender mechanism describes an external application waiting for HTTP requests issued during the scheduling cycle to influence the scheduling decision. Instructing the scheduler to use an Extender in the current implementation of Kubernetes is not limited to a specific profile, but all profiles will use the configured Extender. An Extender can specify the verbs it supports. Verbs in this context refer to filtering, scoring, preempting, and binding.

Note: This work does not replace the KubeScheduler, but deploys a second scheduler configured with an Extender. The second scheduler creates an additional scheduling profile, thus only affecting Pods that are part of the External-Scheduler-Interface. This is done because of the limitation that every scheduling profile will use the Extender, which seems unnecessary in the context of a prototype.

4

Implementation

4.1 Designing the Interface

A Batch Job in the context of the External-Scheduler-Interface is a wrapper around a Distributed Dataflow application. The current version of the interface supports Spark and Flink. The Batch Job resource can also hold any data that an external may want to associate with the job. Listing **??** is an example configuration for the Batch Job resource. The configuration shown is only a shortened version. Complete examples are inside the repository. The Batch Job includes runtime information collected by a profiling scheduler, more details in section 5.2.

The concept of a Testbed allows the reservation of resources inside a cluster. It may be unreasonable to give an external scheduler all available resources in a multi-tenant cluster, thus resources available to the external scheduler can be precisely controlled using Testbeds. Listing 4.2 shows an example configuration for a Testbed resource. The Testbed does not directly specify the final size of the Testbed but rather specifies which Nodes to include and how many slots per node should be created. Finally, it also specifies the size of each slot in terms of a CPU and memory request. The size (both in terms of the number of slots and resources) of Testbeds can only be controlled via the Testbeds manifest. Managing the Testbed is not exposed through the interface, as they are not expected to be changed by an external scheduler, other than using its slots.

The External-Scheduler-Interface allows an external scheduler to control which cluster resources batch applications like Apache Spark and Apache Flink will use for their TaskManager and Executor Pods. No assumptions are made about the Driver and JobManager Pods. Schedulings created by the external scheduler are always directed towards a Testbed. Currently, an active scheduling will claim its Testbed and prevent other Schedulings from using it. The Scheduling resource creates a queue of Batch Jobs that will be run in the specified Testbed. Schedulings currently support a slot-based strategy and a queue-based strategy. The slot-based strategy specifies which job should use which slots, whereas the queue-based strategy only describes a queue of jobs and no specific slots. Listing 4.3 shows an example resource manifest for a Scheduling with a queue of four jobs. Once created (assuming all jobs exist), the interface will

Listing 4.1 Example: Spark BatchJob manifest manifest (shortened)

```
apiVersion: batchjob.gcr.io/v1alpha1
kind: BatchJob
metadata:
  name: spark-crawler
spec:
  externalScheduler:
    profiler:
    - flink-wordcount: 24 (16)
      spark-pi: 27 (16)
      batchjob-sample3: 31 (8)
      #... more information from the external scheduler
  sparkSpec:
    type: Scala
    image: spark-webcrawler:latest
    imagePullPolicy: Always
    mainClass: com.example.webcrawler.WebCrawlerApplication
    mainApplicationFile: "local:///opt/spark/webcrawler.jar"
    arguments:
    - Web_crawler
    - "10000"
    - Kubernetes
    #... more spark specific configuration
```

create the jobs and instruct them to use the first four free slots of the `profiling-testbed` Testbed. Multiple occurrences of the same Batch Job imply creating the job with multiple executors. Since the queue-based strategy does not reference slots directly, an ordering is used to associate an executor with a slot. The queue size is unlimited; jobs that cannot be executed will be queued. Once slots become available again, they will be associated with jobs in the queue. Figure 4.1 demonstrates how a queue that exceeds the number of slots would schedule each executor instance. A job is only submitted if all its executors are submitted because the current

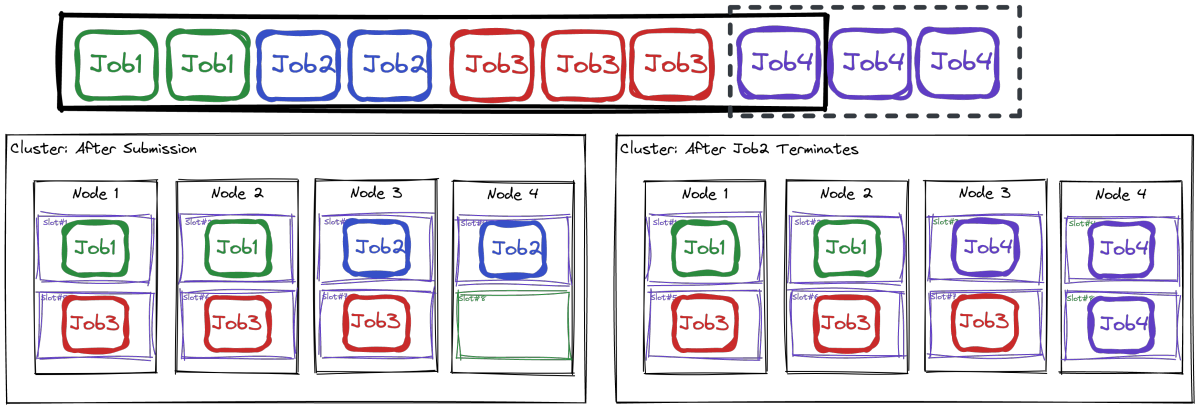


Figure 4.1: Queue with more Jobs than available slots

Listing 4.2 Example: Testbed manifest

```
apiVersion: batchjob.gcr.io/v1alpha1
kind: Testbed
metadata:
  name: profiler-testbed
  namespace: default
spec:
  slotsPerNode: 2
  nodeLabel: "tuberlin.de/node-with-slots-profiler"
  resourcesPerSlot:
    cpu: "700m"
    memory: "896Mi"
```

Listing 4.3 Example: queue based Scheduling manifest

```
apiVersion: batchjob.gcr.io/v1alpha1
kind: Scheduling
metadata:
  name: profiler-scheduling-0
  namespace: default
spec:
  queueBased:
    - name: spark-crawler #0
      namespace: default
    - name: batchjob-spark #1
      namespace: default
    - name: batchjob-spark #2
      namespace: default
    - name: spark-crawler #3
      namespace: default
  testbed: # Testbed the scheduling is targeting
    name: profiler-testbed
    namespace: default
```

implementation can not change the number of executors per job during runtime.

The interface visible to an external scheduler is supposed to be simple. Naturally, an external scheduler could interact with the interface through the Kubernetes API by managing the scheduling manifest. However, a thin HTTP layer is provided if the external scheduler cannot directly access the Kubernetes API. The External-Scheduler-Interface offers endpoints for querying the current cluster situation in the form of the Testbeds slots occupation status and the status of Schedulings and Batch Jobs. The interface provides a REST API that allows the creation and deletion of Schedulings and the ability to update information stored inside the Batch Job manifests. An additional Stomp [**?**] (WebSocket) server is available to not enforce any polling for updates. More concrete information, like Node metrics, can be queried from a metric provider commonly deployed along with the cluster.

The functionality is demonstrated as part of the evaluation, either through the `Manual Scheduler`, which acts as a testing tool and as visualization, and the example scheduler, which uses multiple Testbeds to profile Batch Jobs for its scheduling decisions.

4.2 Architecture

In this section, all components that make up the interface are introduced. Here an architectural overview is presented, and interactions between components are examined.

The current implementation of the External-Scheduler-Interface consists of 5 components that will be introduced in this section but discussed in more detail in the Operator section.

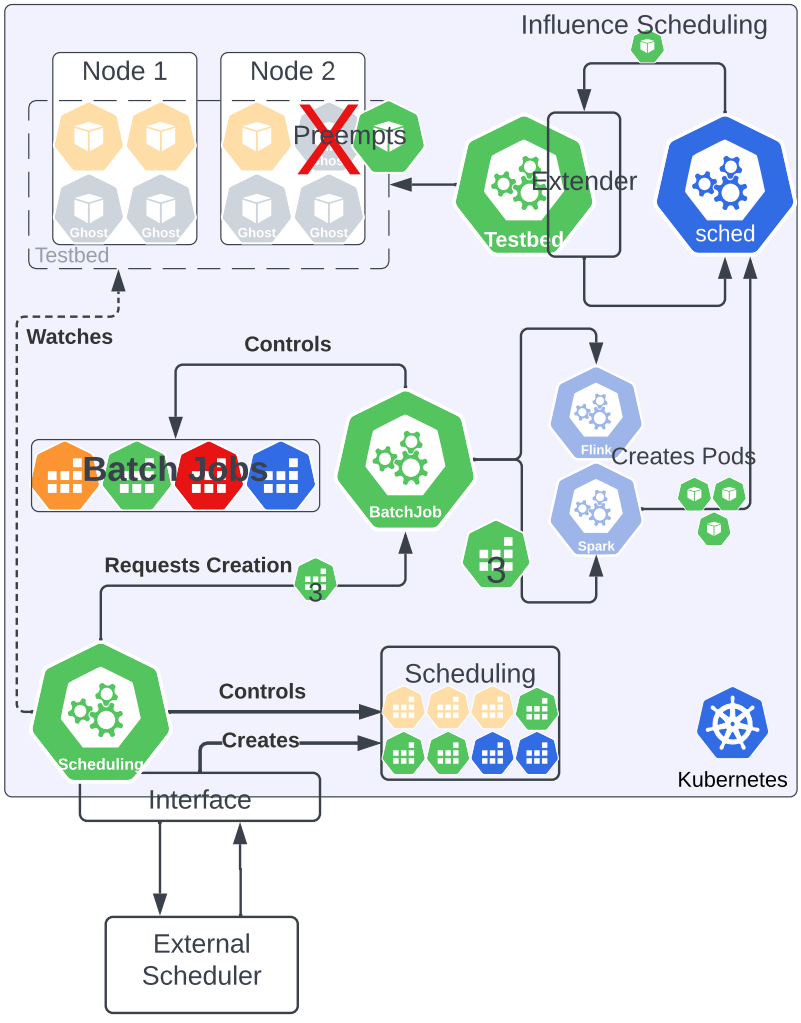


Figure 4.2: Components

The five components consist of three Operators, each managing its resources. The architecture also uses an Extender and the external scheduler facing Web-API.

Additionally to the three reconcilers, three CRDs were designed.

- ◇ **Batch Job** represents an abstract Batch Job application and can store information the external scheduler may want to remember for future invocations
- ◇ **Testbeds** represent a testbed of guaranteed resources available for a Scheduling. A Testbed CR is a collection of slots across the clusters Node that are referenced in a Scheduling
- ◇ **Schedulings** represents the decision done by the external scheduler. A Scheduling maps multiple Batch Jobs to Testbeds available in the cluster. The scheduling also acts as a queue and submits jobs into the slots in order once slots become available.

The Batch Job CR is used to model a reoccurring Batch Job application. To support both Flink and Spark applications, an abstract Batch Job CR is chosen that maps the state of application-specific CR (link SparkApplications[**?**] and FlinkCluster[**?**]) to a common set of possible states. A Batch Job can be claimed by exactly one scheduling. This is because the Batch Job CR models exactly the life cycle of a single application.

Using the extender and preemption, the Testbed reconciler can reserve resources for Pods created by the Batch Job CR. The Slots CR guarantees resources in the cluster by creating Ghost Pods, with a specific resource request representing the size of a slot. The Ghost Pods reserve resources by preventing other Pods requiring scheduling to be scheduled onto the same Node.

Finally, the Scheduling CR is passed to the External-Scheduler-Interface by the external scheduler. Given a set of Batch Jobs and the slots and the Node they exist on, an external scheduler can compute a Scheduling that chooses Batch Jobs and the slots they should run in.

4.3 Operators

In this section, the Operators controlling CRDs introduced in the previous section are discussed.

4.3.1 Batch Job Operator

The Batch Job Operators control-loop is listing for changes regarding the Batch Jobs CRs and applications CRs managed by the Spark Operator and the Flink Operator.

The Batch Job Operator knows how to construct the corresponding application given the Batch Job CRs specification. The lifecycle of the application CR is delegated the the applications Operator. Reusing existing software allows the Batch Job CR to be only a thin wrapper around either a Spark or a Flink specification. In addition to the Spark and Flink CR, it may contain additional information that previous invocations of the external scheduler have stored.

The Batch Job CR requires a user to specify only the required components, like the application image containing the actual application and its arguments like a dataset or where to find it (e.g., using HDFS). Currently, the Batch Job CR only includes a partial application-specific specification. It does not matter if a user submits a fully specified Flink or Spark application because the Operator would overwrite most of the Driver/Executor Pod specific configuration and replication configuration.

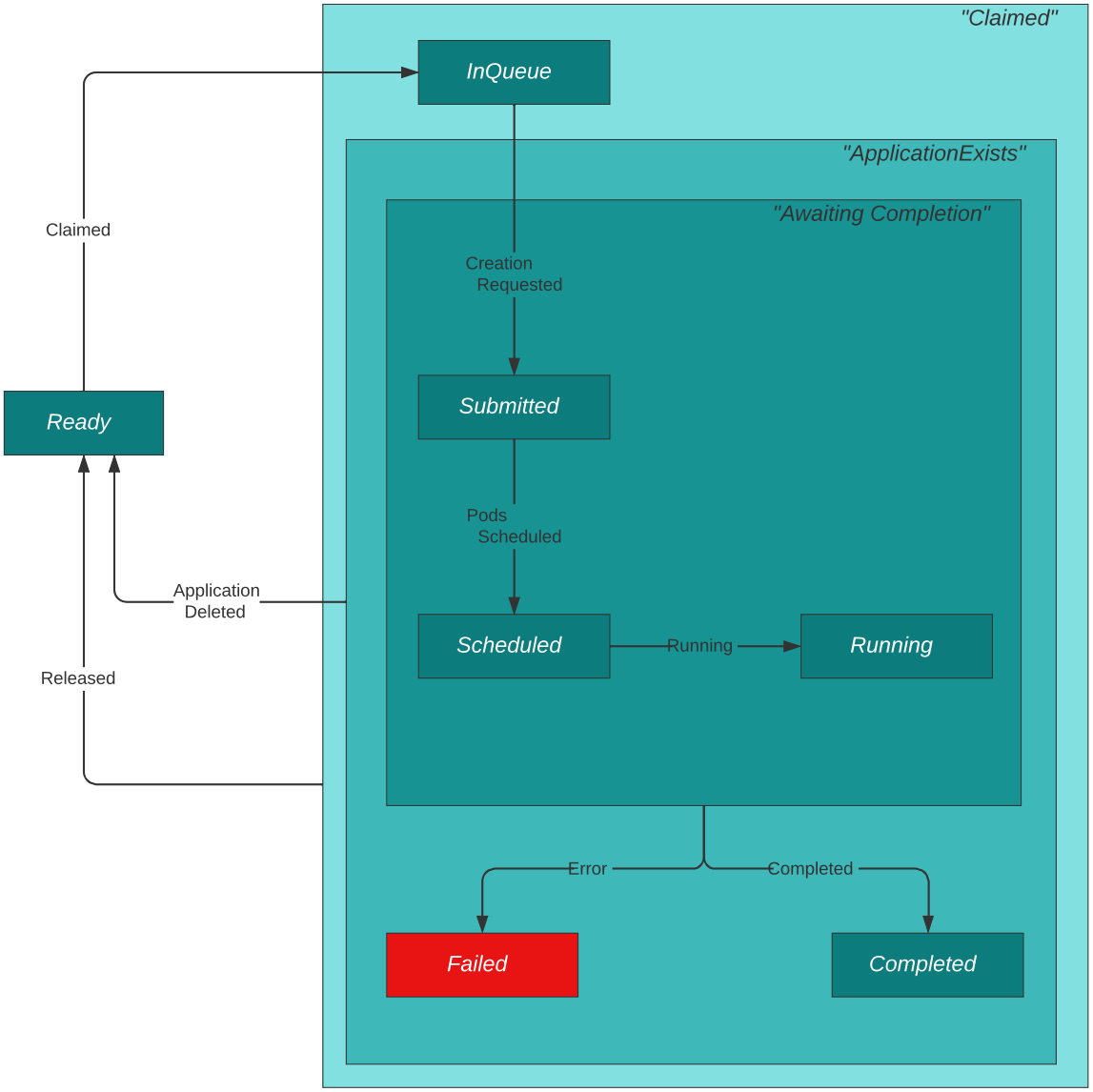


Figure 4.3: StateMachine

Figure 4.3 shows that the Batch Job reconciler is implemented as a nested state machine with anonymous sub-states. The approach was chosen because it creates more comprehensible software, which can be split into components and handle edge cases by design.

Initially, Batch Jobs submitted to the cluster remain in the ready state. While in the ready state, the Batch Job reconciler will not do anything. A Scheduling can acquire a Batch Job. The Batch Job CR will move into the in-queue state until the Scheduling instructs the Batch Job Operator to create the application and track its lifecycle or releases it.

The Batch Job Operator and the Scheduling Operator communicate via the Batch Jobs spec (`.spec.activeScheduling` and `.spec.creationRequest`). If a Scheduling wants to claim a job, it updates the active scheduling spec. This mechanism ensures that

only one Scheduling at a time can use the Batch Job. On the flip side, a Scheduling can claim multiple Batch Jobs. Suppose the active Scheduling releases the job by removing the `activeScheduling` spec, the Batch Job moves back into the ready state. Releasing a job could happen at any time and may even cause any created application to be removed.

Once a Batch Job is in the in-queue state, the Operator waits for the creation request issued by the Scheduling Operator. The request is again done using the Batch Jobs spec and specifies desired replication and the TestBed and slots the application should use.

When configuring the application to be created by the corresponding Operator, there are two types of configuration: configurations specific to the Distributed Dataflow application and configurations specific to the current Scheduling. The application configurations are persisted inside the BatchJob CR. These configurations are used on every invocation of the application and include the program and its arguments. As the name implies, Scheduling-dependent configurations need to be changed depending on the Scheduling and thus must be supplied with the creation request.

After a Batch Job was requested to create the application, application-specific logic is executed. In any case, the actual steps for deploying the applications to the cluster are done by the applications Operator (Flink Operator[**?**] or Spark Operator[**?**]). The Batch Job reconciler only instructs the application Operators with configurations for the Executor/TaskManager Pods, so they are identifiable to the Extender.

When creating the application, the following aspects are configured for the Executor/TaskManager Pods:

Resource Requests: The Testbeds slot size specifies the container resource request. For the Pods to fit inside a slot, they need the correct resource request (currently only CPU and memory).

Testbed Name: The name of the Testbed. Testbeds are not limited to a single Testbed per cluster, so they must be distinguished from another.

Slot IDs: The Scheduling (or the external scheduler) decides which slots are used by which job. For the Executor/TaskManager Pods to be placed into the correct slot (technically the correct Node), Pods need to be identifiable by the scheduler extender.

Replication: The number of Executor/TaskManager Pods depends on the number of slots that will be used for the application.

Priority Class: The Kubernetes scheduler will not trigger preemption unless the application Pods specifies a `.spec.priorityClassName`.

Scheduler Name: The default Kube-Scheduler is not configured to use the Extender. Application pods need to specify the `spec.schedulerName` of the second Kubernetes scheduler, which is configured with the Extender.

Configuration of **Resource Requests**, **Replication** is straightforward, as both the Spark and Flink Operator expose these via their respective CRDs. The Spark Operator exposes the complete PodSpec for Driver and Executor Pods, whereas the Flink Operator only exposes a few PodSpec properties. The Flink Operator had to be extended with the missing configurations. This way **Resource Requests**, **Replication**, **Priority Class**, **Scheduler Name** are configured.

SlotIDs and the **Testbed Name** are not part of the PodSpec, so they are configured using Labels. Labels are a generic list of key/value pairs on every object in Kubernetes.

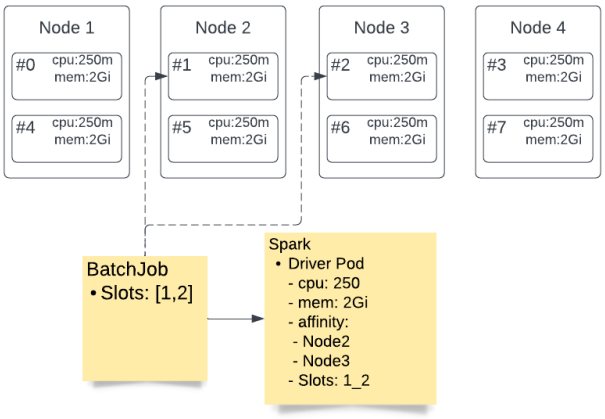


Figure 4.4: Affinities

Ideally, the Batch Job Operator gives every Executor/TaskManager Pod individually a slot ID. The issue is that the application-specific CRs only configure a Pod template. Pods created by the application Operator are managed by a StatefulSet (similar to a ReplicaSet), ensuring the desired replication. This issue is circumvented by leaving the final decision of which Pod goes into which slot to the Extender. The Operator can only configure a list with all **SlotIDs** to the Extender. Figure 4.4 shows how using Affinities limits the set of possible Nodes. Any Node that contains any of the desired slots needs to be considered by the scheduler and the Extender.

Once the application is created, the job moves into the submitted state. It resides there until all Pods where scheduled, at which point it moves on into the running state. The underlying applications state is monitored until it moves into the application-specific completed state (Spark: `Completed` and Flink: `Stopped`). During the implementation, scenarios were encountered in which the Batch Job reconciler was not running. Once restarted, it found applications in a completed state without passing the Scheduling, submission, or running state. To prevent any tight coupling, none of these transitions are required to be considered a successful execution.

The Batch Job reconciler tracks the time an application ran by creating timestamps once it started running and its completion.

4.3.2 TestBed Operator

The TestBed Operator monitors changes to Pods and Nodes in addition to controlling Testbed CRs. The TestBeds CR is supposed to model a collection of slots located in a cluster of machines. Slots can have specified resources. While no application is running inside a slot, it is considered *free*. To reserve resources in the cluster and thus guarantee applications supposed to be deployed inside a *free* slot actually to get the resources, the Testbed Operator needs to:

- **Reserve Resources** by using so-called `Ghost Pods` inside the cluster that specify a resource request and thus reserve the resources
- **Preempt** Ghost Pods for Pods that wants to be deployed inside a slot

The Testbed reconciler listens to changes to the Testbed CR and the current cluster situation. It ensures that the correct number of Pods with the specified resource requests are always deployed onto the cluster. The Testbed CR is composed of the following configurations:

- ◇ Label Name to identify any Nodes that form the Testbed. Only the label name is specified, not a specific value. The value is later used to create a distinct order of slots in the cluster.
- ◇ Number of slots per Node
- ◇ Resource request per slot

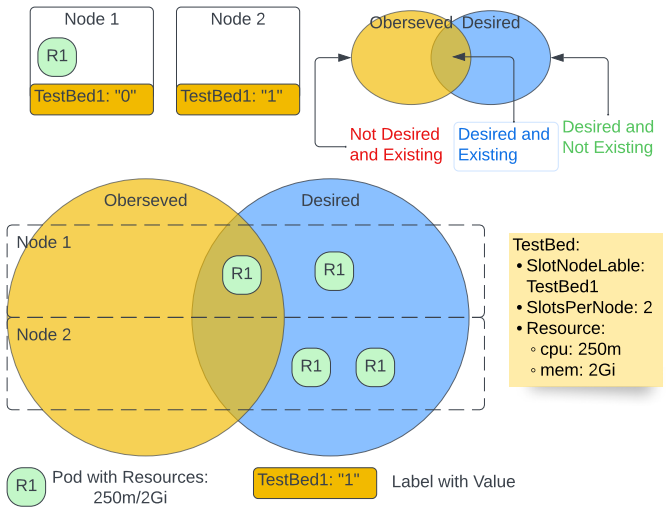
Given the Testbeds specification, the reconciler listens for all changes to Nodes **with** the specified label. It also needs to listen for Nodes **without** any label if the label was removed and the Testbed needs to be resized. Further, it also listens to changes to any Pod part of the Testbed.

The typical reconciliation loop works as follows:

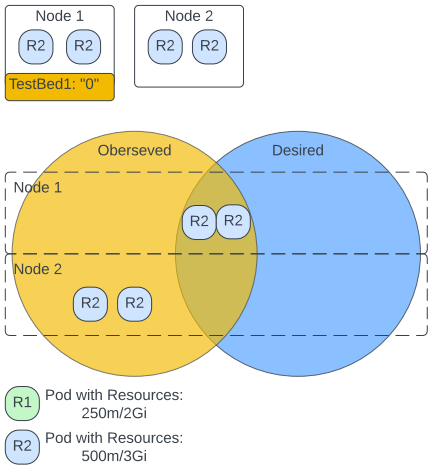
- ◇ Fetch the current cluster situation
- ◇ Calculate the desired cluster situation
- ◇ Find the difference. Either delete undesired Pods or create desired Pods

Fetch the current cluster situation by fetching all Pods with the **SLOT** label. Pods are then grouped by their Node, thus creating a list of Pods per Node. The desired state is calculated by modeling Pods for every slot and grouping them by Nodes. When comparing Pods, we consider them equal if they reside on the same Node, have the same resource request, and have the same *SlotPositionOnNode*.

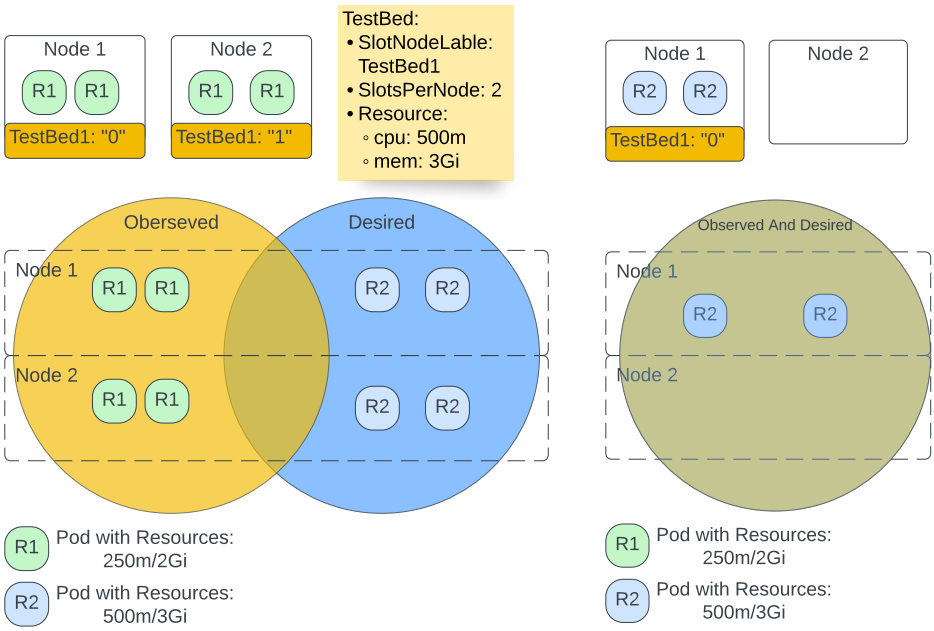
Note: The position of slots on a Node does not matter because slots on are Node are only a logical abstraction.



(a) New Pods need to be created



(b) Node Change: Pods need to be deleted



(c) ResourcePerSlot Change: New Pods need to be created (d) Desired State: No change

Figure 4.6: TestBed Observed and Desired

The reconciler now builds a set of observed Pods and a set of desired Pods. 4.5a shows an example scenario where the control loop realizes that Pods from the desired state are not in the current state, thus creating the missing Pods in the *desired and not existing* set. In a different scenario displayed by 4.5b the label on a Node was removed, thus reducing the number of slots inside the Testbed. Pods that are in the *existing and not desired* set will be removed. The final set is the *desired and existing* set, which contains Pods that already have the correct resources requirement and are placed on the correct Node.

Currently, the SlotOccupationStatus holds the following information:

- **NodeID** and **NodeName**: which is derived from the Test-Bed Selector Label on the Node
- **Position**: which is the SlotID,
- **slotPositionOnNode**: where the position does unique among the whole Test Bed, SlotPosition on Node is only unique per Node
- **PodName** and **PodUID**: The Name and the Unique Identifier of a Pod that is currently residing inside the slot
- **state**: is the current state of the slot, which can either be *free*, *reserved*, or *occupied*

4.3.3 Extender

Extender Component is integrated within the TestBed Reconciler. If the reconciler detects that the cluster is in progress, it pauses its control loop. This prevents both components from acting on the same Testbed concurrently, thus preventing any unexpected changes to the Testbed slot occupation state while the Extender looks for free slots. Currently, a cluster is considered in

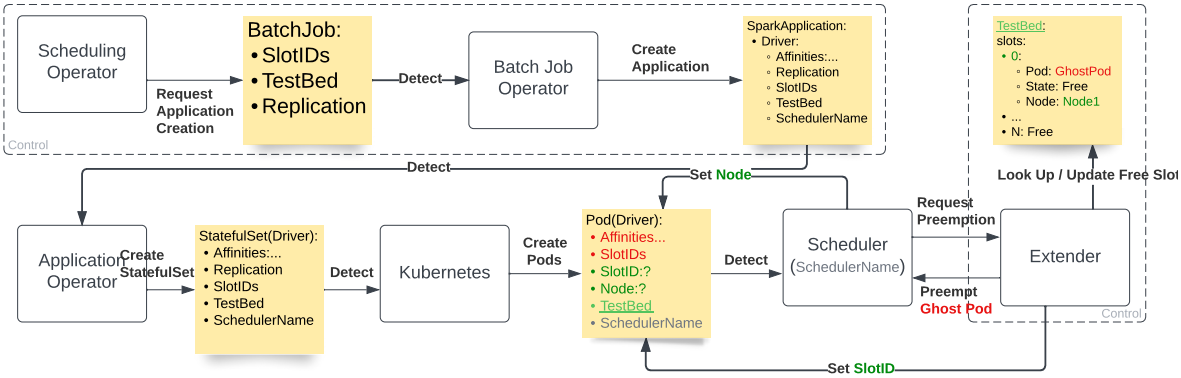


Figure 4.7: Components under control of the External-Interface System

progress if any Pods require Scheduling (.spec.NodeName is not set) or are terminating (deletion timestamp is set).

The Extender is the component that directly interacts with the Kube-Scheduler. An additional scheduler, with an additional scheduling profile, is running concurrently to the default Kube-Scheduler. The custom scheduler (referred to as Kube-Scheduler) is configured to use the Extender. To guarantee the Scheduling of Pods onto the TestBeds slots, the Extender extends the Filter and Preemption extension points of the Kubernetes scheduling cycle. The main problem the Extender can solve is that the Batch Job Operator does not have full control of Pods created downstream by the applications Operator.

Figure 4.7 shows which of the components and resources managed by them are under the control of the External-Interfaces System. The Batch Job Operator can only control the Application CR created. The Application CR only describes a single PodSpec, which will later be replicated into multiple Pods by the replication controller, part of the StatefulSet. Thus, it is impossible to set Pod-specific configurations, like the SlotID, at the Batch Job Operator Level. However, with enough information, all Pods can be configured for the Extender to figure out which Pod belongs in which slot.

Note: PodSpec here only refers to the TaskManager/Executor PodSpec, as the External-Interface does not handle Scheduling of the JobManager/Driver Pods.

Figure 4.8 shows how the Kube-Scheduler is influenced to schedule Pods onto Nodes with the correct slot. The number of possible Nodes is first limited by all Nodes containing any of the slots using affinities. Finally, the Extender chooses the right Node with the designated slot. The first step is already implemented inside the Kube-Scheduler. The second step is elaborated in more detail.

During development, multiple scenarios of interaction between Kube-Scheduler and Extender were identified.

If the Kube-Scheduler detects that neither of the possible Nodes has enough resources available, it will immediately trigger preemption, thus forgoing the Filter-Extender. If no preemption is required, the Filter-Extender is queried to limit the possible nodes further. The Filter-Extender

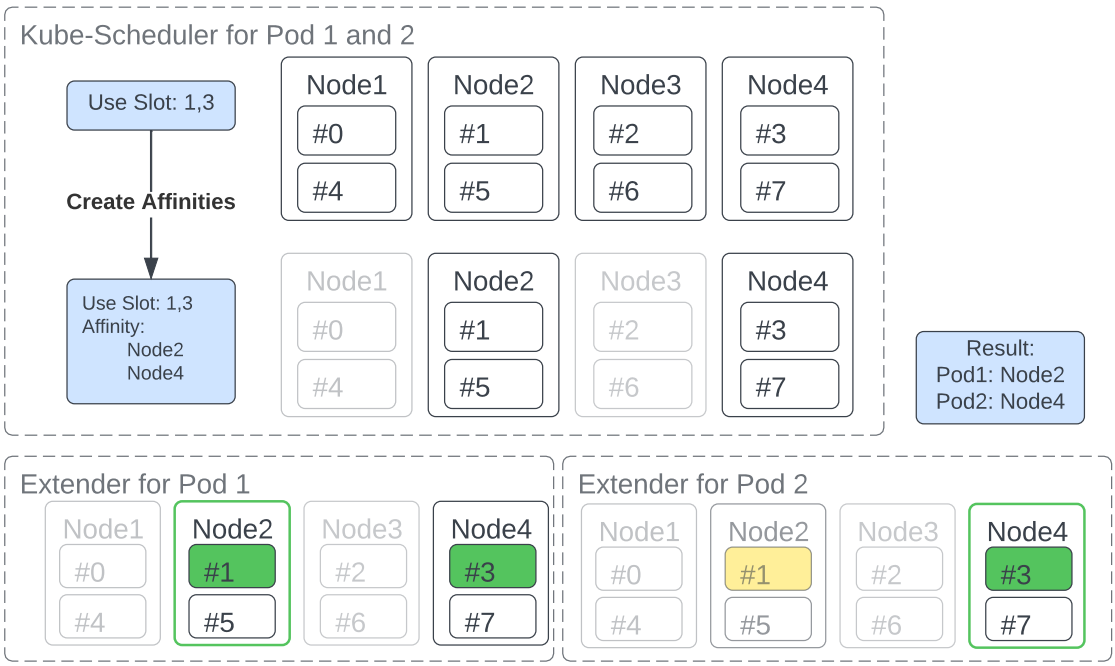


Figure 4.8: Kube-Scheduler limits Nodes, Extender selects Node with slot

arguments only contain Nodes that match the resource requests requirement of the Pod. If the Kube-Scheduler offers Nodes that the Extender already used for other Pods, the Filter-Extender limits the possible Nodes to an empty set, triggering preemption.

The Kube-Scheduler has its internal default preemption algorithm. It simulates scenarios of preempting pods with a lower priority until any Node passes its filters (usually the resource request filter). If the internal preemption does not find a possible scenario, preemption is canceled, and the Pod becomes unscheduable. This scenario skips the Preemption-Extender entirely and creates a problem for the External-Scheduler-Interface system. Pods created downstream by the Batch Job Operator are configured with a higher priority than Ghost Pods to prevent the scenario from occurring, thus guaranteeing that preemption will always query the Preemption-Extender.

The Preemption-Extenders arguments include the preemptor Pod and possible preemptees. Because internally, the Kube-Scheduler stops when finding the first potential victim for preemption, the preemptees are only a suggestion and are ignored by the Preemption-Extender.

The Preemption-Extender uses the **Slot-IDs** and the **Testbed Name** that the Batch Job Operator placed on the Pod. It first searches through all desired slots of the Testbed to find slots that have already been reserved for the Pod. If no Slot has been reserved yet, the Extender finds the next possible free slot. A scenario where no slot is found should not happen because the controlling Scheduling will only submit new jobs once enough slots become available.

Since the Kube-Scheduler could invoke the Extender multiple times for the same Pod, the first invocation reserves a Slot, and subsequent requests will always return the same slot. Slots are marked as reserved using the Testbed CR. Because the Testbed is based around preempting

Ghost Pods that reserve system resources, both the Filter-Extender and the Preemption-Extender will prepare preemption internally. At the end of either scenario, the Pod that requires scheduling will have its **Slot-ID** set, and the Ghost Pod previously residing inside the slot is either marked as preempted or removed. The Pod will also be marked as **NonGhostPod**, so the Testbed Operator can preempt the Ghost Pod with the same Slot-ID if it was not deleted through the Kube-Schedulers preemption.

4.3.4 Scheduling Operator

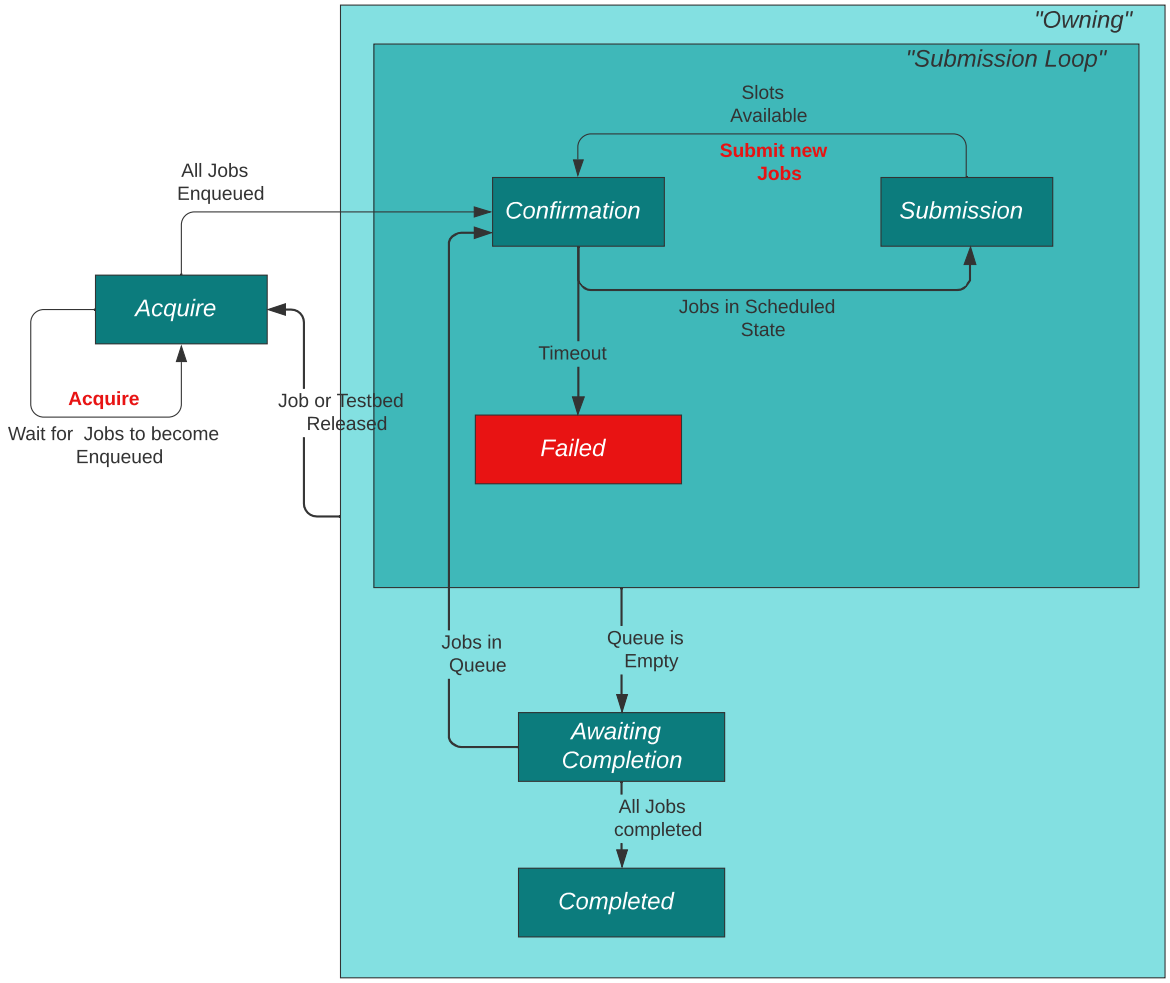


Figure 4.9: Scheduling state machine

Like the Batch Job reconciler, the Scheduling Reconciler is implemented using a nested state machine (Figure 4.9). In addition to changes to Scheduling resources, the reconciliation loop is also triggered on changes to Batch Jobs or Testbeds.

The Scheduling CR contains a collection of jobs, a slot selection strategy, the target Testbed. The Scheduling tracks the execution of all its jobs and submits new Jobs once old jobs have finished and slots become available again.

Initially, the Scheduling was planned only to support offline Scheduling, where an external scheduler plans the execution of multiple jobs in advance. However, in theory, updating the Scheduling spec would allow online scheduling. Still, it is rather unreliable in the current state, as it only allows jobs to be added to the end of the queue.

Slot selection strategies do not aim to provide a full scheduling algorithm. They are just means for an external scheduler to describe which Job should use which slot.

The lifecycle of a scheduling is described using a state machine. Once the Scheduling CR is created, it moves into the Acquire state. While in the Acquire state, the scheduling tries to acquire all its jobs and its Testbed. If it detects that any of them is currently in use, it will release all of them to prevent any deadlocks. It remains in the acquire state until all acquired Batch Jobs move into the InQueue State.

The core submission loop of the Scheduling reconciler moves from the Confirmation state to the Submission State until the queue becomes empty.

Two distinct states are needed between submissions of jobs because the Testbeds slot occupation status is not updated immediately. The Confirmation state awaits previously submitted jobs from confirming their scheduling (Batch Job in Scheduled state). Once previous submissions are no longer pending, the Scheduling waits for slots to become available.

Depending on the Strategy, either specific slots or just a number of slots need to be available to submit the next job from the queue.

Once the queue becomes empty, the core submission loop is exited, and the Scheduling moves into the AwaitCompletion. If all jobs are completed, Scheduling is completed.

All jobs and Testbeds are released once the scheduling is deleted.

The Schedulings Resource may be modified at any time, moving the scheduling from the AwaitingCompletion state back to the Core submission Loop.

4.3.5 External-Scheduler-Interface

The external-scheduler-facing interface is currently part of the Scheduling Reconciler. Most of the functionality has already been discussed in the previous chapter.

The interface is only a very thin wrapper around the Kubernetes API. In its current state, it is only used to hide unnecessary complexity from the external scheduler and offers are more simplistic API to interact with Kubernetes and the External-Scheduler-Interface components.

A reference implementation for using the API is part of the Manual Schedulers website.

5

Evaluation

5.1 Testing

The functionality of the External-Scheduler-Interface is tested using a combination of Unit Tests and Integration Tests.

Unit tests are highly specific towards smaller system components and thus will not fit the written thesis's scope. They are located within the repository (Appendix).

Integration Tests aim to test the bigger picture. Testing all functionalities is not feasible in a system composed of many distributed processes, as setup would require recreating a cluster with all its software components. However, testing the complete system with all its components is possible with an already established cluster. The Manual scheduler aims to ease the use of External-Scheduler-Interface.

For the integration tests to run in a timely manner, a common practice is to *mock* Software components, which are either not under immediate control or are very expensive to set up. The Java Operator SDK and the Fabric8 Kubernetes client can Mock the Kubernetes Cluster. Mocking the entirety of Kubernetes can not be accomplished, but the Kube-API-Server itself is enough to test the Operators. With the Kubernetes API-Server mocked, we can simulate the abstract state (declared state) and verify that changes to the abstract state trigger the correct actions by the Reconcilers.

Overview of Software Components used in the Integration Tests:

- ◇ Kubernetes API: For testing purposes, the Kubernetes API is just a CRUD REST application that can create, read, update, and delete resources. The Fabric8 Kubernetes Client offers a `KubernetesMockServer` with sufficient capabilities to test the Operators.
- ◇ Kubernetes Scheduler: Pods submitted to the Kubernetes API will be treated like regular resources and thus will not be scheduled onto Nodes. The Kubernetes API does not include any controllers for Kubernetes native resources. A simple implementation for a

Kubernetes Scheduler has been implemented to test the functionalities of the Extender.

- ◇ Application Operator: Both of the application Operators are implemented in Go. Integrating them into the integration tests is out of the scope of this work, if not even unreasonable. Applications supposedly created by the application Operators are managed programmatically throughout the test cases.
- ◇ Batch Job, Scheduling, and Testbed Operator: The controller and the generated CRDs are used for the integration tests. Controllers are configured to use the `KubernetesMockServer`.

Basic functionality of the Reconciler components, at least on the resources manifests level, can be verified using the integration tests.

The Manual Scheduler Frontend can be used to test the External-Scheduler-Interface on an established Cluster with all components installed.

Note: At the time of writing the Fabric8 Kubernetes Server Mock does not appear to be fully thread safe which results in flaky tests. With the version 6.0.0 concurrency issues seem to be resolved.

5.1.1 Manual Scheduler

The Manual Scheduler is a web frontend that interacts with the interface and allows a user to create a scheduling and test it. The Frontend visualizes the current state of Testbeds, Schedulings, and Jobs.

5.2 Example Scheduling Algorithm

The Interfaces usability is evaluated with an exemplary implementation of a non-trivial scheduling algorithm. Since the Interfaces can manage multiple TestBeds inside the same cluster, a Profiling-Scheduler approach is chosen to highlight some of the interface's features.

The example scheduling algorithm is used on a 5 Node cluster running inside the Google Cloud Platforms Kubernetes Engine (GKE). During development, smaller Nodes with a single vCPU and 8Gi of Memory were sufficient, but for the final evaluation, Nodes were doubled in capacity.

Two Test-Beds are created using the Test-Bed CRD, the Profiling TestBed, and the TestBed for the actual execution of Jobs. The slot size was chosen depending on the resources available. For the final evaluation, a slot size of 750m CPU and 6Gi Memory leaves enough resources available for the cluster's control plane, managing Pods (Driver, JobManager), and the Operators running inside the cluster.

Contrary to the Scheduler Thread, which only creates a scheduling if requested (via stdin), the Profiler Thread runs at all times, updating and refining the Co-Location Matrix by choosing job pairings with the least data points.

Co-Locations are described as a simple runtime in seconds. By iterating over the available job pairings, the Profiler builds a Co-Location Matrix. The Profiler creates a cumulative moving average for each job pairing.

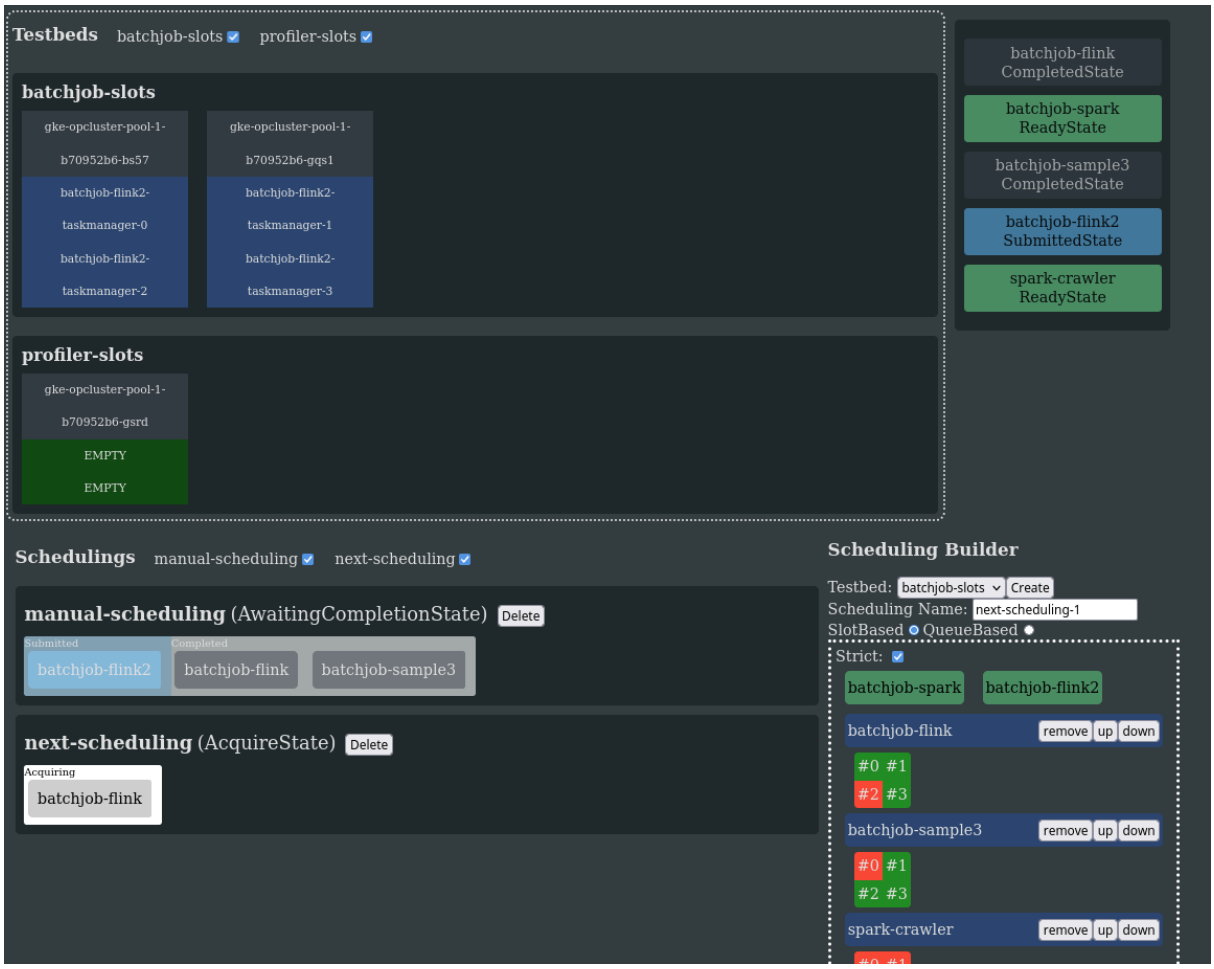


Figure 5.1: Manual Scheduler

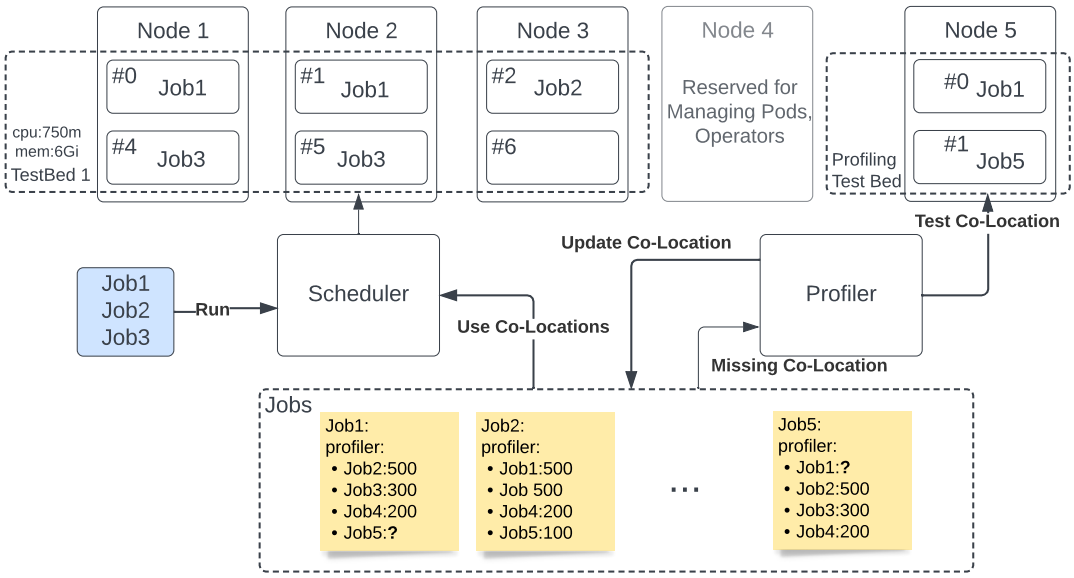


Figure 5.2: Architecture of the Example Profiler-Scheduler

Jobs cannot be paired with themself since the acquire/release mechanism only allows a single execution per job at a time. In theory, Jobs can be Co-Located with itself by deploying the application with a replication of two, but this could not be directly compared to Co-Location with a different job, as work done is split between both instances.

Note: The matrix is not symmetrical because the runtime of each job is used, not the runtime of both jobs (or the runtime of the complete scheduling). The runtime of the scheduling is the time after acquisition until all jobs have been completed. The Batch Job Operator only tracks the application's time inside the running state. This approach was chosen because applications may have vastly different startup times, which will become insignificant for long-running jobs.

The Scheduling Thread is the traditional scheduler. Given a list of Jobs, the Scheduling Thread tries to find optimal scheduling regarding total runtime. The scheduler takes a greedy approach choosing the co-located job based on the job with the shortest runtime to keep the evaluation simple. Replication of each job is selected based on the number of slots (Replication = NumberOfSlots / NumberOfJobs). Empty slots are again greedily filled with jobs suited best for co-location, not allowing a job to be chosen more than once.

Both the Profiler and the Scheduler run in parallel. If any of them cannot acquire their jobs, the scheduling will wait until they become available.

5.3 Limitations

The prototype implementation of the External-Scheduler-Interface comes along with limitations. This section will outline a few of them. The final Future Work Section will be a continuation of this section. This section focuses more on limitations and problems encountered during Testing. The Future Work section picks up on features that were not feasible for a prototype in the given time frame.

One of the biggest problems, which created the most trouble, is the locking Mechanism. Locking is required because concurrent use of the Testbed is unreliable. The problem is that slot reservations do not happen instantaneously but rather after the pods are scheduled, leaving the Testbed in a seemingly empty state, wherein reality slots have already been used by a Scheduling creating batch jobs. This also leads to a somewhat awkward design of the Schedulings state machine. Scheduling has to wait for confirmation of its previous scheduling cycle before the Testbeds slot occupation status is updated. The locking mechanism also creates the drawback of potential deadlocks. Finalizers are used to release claimed jobs and testbeds once Scheduling is deleted.

Another commonly encountered problem is dealing with errors. Due to poorly written software or unexpected cluster changes, errors were frequently encountered. One of Kubernetes' core aspects is self-healing, where control loops move the cluster from a damaged state back into a healthy state. Kubernetes native resources like Jobs have policies for automatic restarting. On the Interfaces side, error detection is handled by the resources state machine. The state machine approach is supposed to eliminate frequent checks for preconditions. If a precondition fails, the resource moves into a failed state. The current implementation does not handle failures particularly well, and Batch Jobs encountering errors will remain in the failed state until recreated.

5.4 Discussion

6

State of the Art

6.1 Volcano

Volcano is a System Batch-Job Scheduler made for High-Performance Workloads on Kubernetes. Volcano extends Kubernetes with functionalities that Kubernetes do not natively support. Some of these functionalities are critical when working with High-Performance Workloads, like "PodGroups". In a scenario where a Framework might want to create multiple Pods for its computation, the resources inside the cluster only allow for a few of them to be deployed. Applications could encounter deadlocks, requiring more Pods to be deployed to progress. The Concept of PodGroups prevents Pods from being scheduled unless all of them can be scheduled.

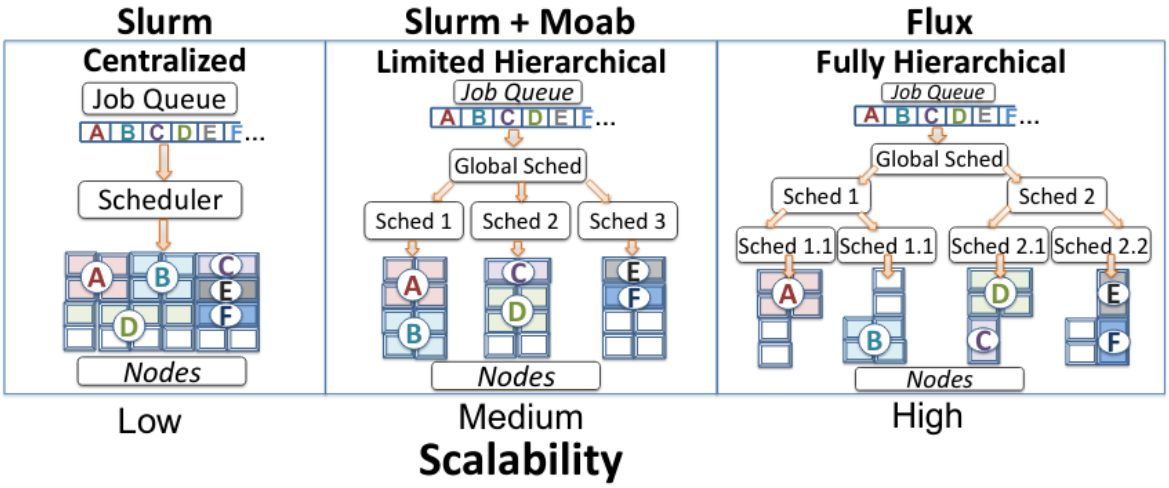
The Volcano scheduler is based on the Kubernetes Scheduling Framework, influencing the scheduling cycle at the extension points. This way, Volcano can implement many scheduling policies, which High-Performance Batch Applications commonly use. Volcano is more concerned with policies around the actual scheduling algorithm. In contrast, the External-Scheduling-Interface, introduced in this work, focuses on aiding the development of the actual scheduling algorithm. In theory, algorithms could be implemented using Volcano, but since Volcano provides just a thin layer above the Kubernetes Scheduling Framework, using Volcano for the development of new scheduling algorithms does not seem like a plausible choice.

With both frameworks supporting Batch Scheduling in different ways, the External-Scheduling-Interface and Volcano could complement each other, but there has been no further investigation.

6.2 Flux

Flux aims to find a solution for Converged Computing, a Term used, when describing the move of traditional HPC computing to the Cloud Native Computing Model. HPC Systems traditionally bring high performance and efficiency due to sophisticated scheduling, where the Cloud offers Resiliency, elasticity, portability and manageability. Traditional HPC Batch Scheduler, will not keep up with the growth of systems enabled by the cloud. Established HPC frameworks,

such as Slurm[**?**], use a centralized Scheduler. Flux Identifies scalability issues, that the Scheduler is limited, in the maximum number of jobs it can handle. To Prevent the scheduler from overwhelming, job submission needs to be throttled, which will decrease job throughput. While not strictly related to scheduling, a centralized approach, will also fail at tracking the status of jobs running inside larger clusters. Sli



Flux introduces a new HPC Scheduling model to address the challenges, by using one common resource and job management framework at both system and application levels. Using an Hierarchical Scheduler applying the divide-and-conquer approach to scheduling in a large cluster. The hierarchical scheduling model, enables jobs to create their own schedulers, which is used to schedule its sub-jobs.

Another approach the Flux Scheduler takes, to scale with the increasing number of jobs in a cluster, is to aggregate jobs, that are similar and arrive within the same time-frame, into single larger job.

7

Conclusion and Future Work

7.1 Conclusion

Kubernetes extendible design, using the Operator pattern and extending the Kubernetes Scheduler, allows the External-Scheduler-Interface to offer a simplistic Interface to scheduling of Distributed Dataflow Application on a Kubernetes Cluster. Using the interface allows an external scheduler to control Batch Job applications' location without a deeper understanding of Kubernetes. Research in scheduling algorithms can build on top of the interface by creating controlled test scenarios using Testbeds. Commonly tested Batch Jobs, which are already configured and prepared, can be shared among researchers. Previously existing scheduling Algorithms like Mary and Hugo[**?**] can now solely focus on implementing the Scheduling Algorithm and can be less concerned with configuring a cluster.

Although far from perfect, the prototype has been evaluated to run many scheduling cycles, for hours, consistently without failure. In its current state, it requires polishing regarding error reporting and gracefully handling of errors.

7.2 Future Work

The intended use-case for the Interface is the development of new scheduling algorithms. Arbitrary-sized Testbeds can span across a single cluster controlled by Kubernetes. Kubernetes has been shown to scale with large clusters, and with Kubernetes v1.23 clusters with up to 5000 Nodes are supported[**?**]. During the prototype development, a cluster with only a few Nodes was used, thus leaving the scalability of the Extern-Scheduler-Interface in an uncertain state. It is unlikely for a single Testbed to overwhelm the Operator due to limiting the access to *Testbeds* to a single *Scheduling*. Furthermore, Batch Jobs are rather long-living and do not require much maintenance (at least from the Interfaces perspective). However, there are currently no limitations on how many *Testbeds* may exist in the same cluster, and thus the Interface may be overwhelmed with too many concurrent *Testbeds*. Currently, the Interface has

no leader election mechanism and therefore cannot scale horizontally. Using multiple Operators in different namespaces does not require a leader election mechanism but would require refining the use of namespaces inside the Interface.

A Namespace in Kubernetes is the mechanism that isolates groups of resources within the same cluster. In a shared cluster, it is commonly used to not interfere with other users without revolving back to statically partitioned clusters and lose the benefit of resource sharing. The current implementation only supports limited use of namespaces since it was developed on a private cluster and could use the "default" namespace for all its components. The Integration Tests use a different namespace, and the Interface is not tied to a specific namespace. However, for now, all components run inside the same namespace. This leaves the question if actions done by the Operator should be limited to a single namespace or if it should be able to interact with resources in every namespace. Furthermore, this question is not just limited to the Operators, which are part of the prototype but also application-specific Operators. The Spark-Operator currently allows both options, either watch all namespaces or limit the scope to only a single namespace.

The Interface currently does not expose the namespace to the External-Scheduler, because every resource must be inside the Interfaces namespace. This further promotes limiting the scope per instance of the Interface to a single namespace and thus limiting required privileges. However, some actions required by the Interface need access to resources across all namespaces. First of all, the *Testbed Operator*requires access to the Nodes (not namespaced) and requires access to all pods running in every namespace to calculate the available resources before creating a *Testbed*. The issue with Nodes not being limited by a namespace is that multiple *Testbeds* may use the same Node, which would undoubtedly cause trouble with the current implementation.

The number of executors can not be changed at runtime. In the context of online scheduling, an external scheduler may decide that rescheduling executors across the testbed might be beneficial, especially for longer-running jobs, where the overhead will be less significant. In theory, the External-Scheduler-Interface supports online scheduling. However, updating the manifest will not change any existing jobs.

The last limitation that would prevent the Interface from being used in a *production*ish environment is the lack of Security. Both in terms of malicious Users (or just forgetful), who can reserve cluster resources with a Testbed, and accessing the cluster from outside via the API Endpoints, without proper authentication and authorization. The first issue is hard to circumvent because a malicious user could do that anyway, and Kubernetes Resource Quotas[**?**] can limit the resources per namespace, but they would also limit a non-malicious User. Automatically resizing a Testbed if it is not in use for a while could prevent unnecessary reservation of resources and defeat the purpose of reserving resources for an External-Scheduler. The Second issue requires API-Endpoints not to be publicly available. During development, the Interface was accessed using port-forwarding, which requires authorization to the Kubernetes cluster. For future work relying on port-forwarding seems plausible. However, creating an actual External-Scheduler down the line would require proper Ingress configuration to the cluster and a security model.