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 4Gi of Memory were sufficient, but for the final evaluation, Nodes were doubled in capacity.

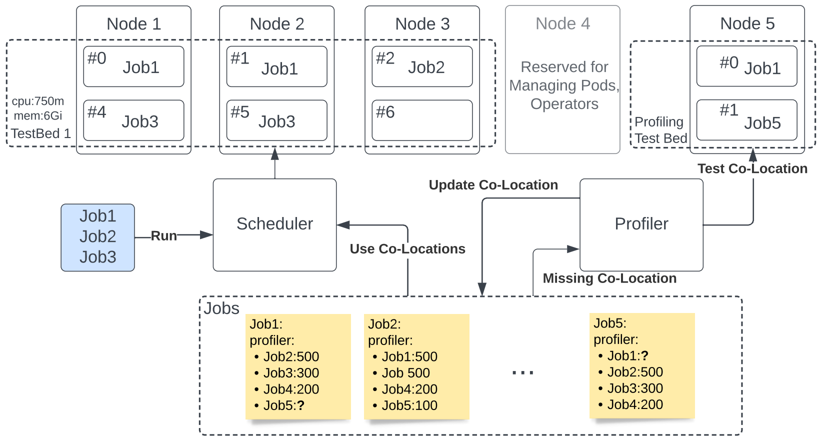

Figure 1: Architecture of the Example Profiler-Scheduler

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 3Gi 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. To test the systems stability the scheduling thread was later modified to create schedulings by choosing jobs at random.

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.

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.

	batchjob-flink	batchjob-spark	batchjob-sample3	batchjob-flink2	spark-crawler
batchjob-flink	/	36 (5)	28 (11)	24 (16)	34 (11)
batchjob-spark	70 (4)	/	93 (4)	68 (4)	94 (4)
batchjob-sample3	444 (4)	480 (6)	/	450 (3)	493 (3)
batchjob-flink2	24 (16)	36 (5)	30 (11)	/	25 (10)
spark-crawler	173 (3)	203 (5)	211 (3)	171 (3)	/

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.