A System Analysis to Dkron Scheduler

Miguel Albuquerque, 1105828[1][1105828]

Instituto Superior Técnico, Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal
`miguel.albuquerque@tecnico.ulisboa.pt`

Abstract. Dkron is a golang written distributed cron job scheduler that is fault-tolerant (Reliable), and allows for concurrent job execution. In this project... Summary Method? Summary Results?

Repository: github.com/miguel-msa/dkron-study
Commit hash: **PUT COMMIT HASH HERE**
Video link (youtube): **https://youtu.be/SxbSg0Dx16s**

Keywords: Dkron · Distributed · Benchmark · Scalability · Consensus · Dkron Server and Dkron Agent · Raft.

1 Introduction

Distributed systems, in simple terms, are a group of systems or processes that work together for a common goal. Being distributed, some particularities need to be considered, for instance how do they interact, and how do they agree on a decision. Agreeableness in distributed systems is a concept of grand study, where we have examples like Paxos and Raft, being both consensus algorithms.

A job scheduling system purpose is to automate execution of jobs, do repetitive tasks, and when well-defined and organized, automate entire workflows. It is expected that these jobs run within defined conditions, for instance time-based conditions. Other use cases are resource management (e.g., scheduled to execute for off-peak hours) and avoid manual intervention.

Dkron is a distributed job scheduling system that on the surface:

1. Has flexible tag-based job definition
2. Allows for Distributed execution, where these jobs can be ran in a distributed mode.
3. Has no Single Point of Failure.
4. Easy to deploy with built-in replicated storage (BuntDB) relying on the Raft protocol.
5. Provides a Web-GUI for administration.

The above characteristics, and more, are also explored in 2

When selecting the system to analyse, some of the factors driving the choice were:

1. Interest in exploring Go source code.
2. Explore a system that is not a database

From the options, Dkron raised a question: *What is the performance of a scheduler, seemingly, focused on availability and reliability*

As this is not thoroughly explored by Dkron, I got interested in this system and how it would scale for reliability.

Ultimately, Dkron is a golang written scheduler with specific particularities for some key use-cases. Its claims are not bold, neither on its performance, nor on how it actually does scheduling. This opens an opportunity to not only explore something unclaimed, but also on how Dkron found the optimal point, if exists, between performance and reliability.

The analysis will consider these use-cases and apply a ...benchmark... to analyse the performance and infer Dkron's performance on workloads similar, as in trying to represent, to such use-cases.

2 System Description

Further exploring Dkron's characteristics from the summary in the introduction, some of Dkron's particularities are:

1. **Concurrency:** A job can be either ran concurrently, by default, allowing for overlapping jobs, or forbid concurrency.
2. **Cron spec support:** A job can be defined either by using Dkron's predefined schedules, which are more. declarative, e.g., *@hourly*, or using cron expression format.
3. **Executors:** Plugins to execute the jobs, e.g., Shell to run the job in the current node, or HTTP to send a request.
4. **Metrics:** Dkron can send metrics to Statsd and provide prometheus format metrics.
5. **Job Retries:** A job can be declared to retry in case of failure.
6. **Embedded Storage:** Dkron uses an embedded KV store based on BuntDB.
7. **State Storage:** The scheduler state is replicated between all server nodes using Serf.
8. **Target nodes spec:** By default every node will run a job, however there is the ability to set tags for jobs to be ran only by target nodes identified with those same tags.

9. **Clustering:** Clustering in Dkron provides fault tolerance when the quorum is equal or higher than 3.
10. **Leader Recovery:** Detects if the leader is missing in the work pool, and (usually) elects a new leader.

Dkron has 2 kinds of nodes: Server and Agent. Both are cluster members available to run scheduled jobs

An agent can handle job executions, run scripts and return the resulting output to the server.

A server does everything an Agent does, plus: schedules jobs, handles data storage, participate on leader (Raft) election, dispatches the execution of jobs either to Agents, or other Servers.

A Dkron cluster has a leader, for Raft based distribution, which is responsible to start job execution queries in the cluster.

By default, all nodes execute each job. This can be controlled through the use of tags and a count of target nodes having this tag. This allows to run jobs across a cluster of any size and with any combination of machines needed.

All execution responses are gathered by the scheduler and stored in the BuntDB storage.

```
{
    "name": "job_name",
    "command": "/bin/true",
    "schedule": "@every 2m",
    "tags": {
        "my_role": "web:1"
    }
}
```

Above is an example of a job that should run only on one node, in any that has the value *web* for the tag *my_ role*

– Dkron can either run on a single node (by default) or on a cluster with multiple nodes.

CHECK: Create a Job @ https://dkron.io/docs/basics/getting-started more details there might be interesting

2.1 No Single Point of Failure

By leveraging Raft consensus algorithm, this no Single Point of Failure characteristic is very relevant for systems that, e.g., depend on other running jobs or trigger to function - the scheduler. The System using these schedulers might be fault-tolerant, but the scheduler itself might not, making these "fault-tolerant" systems, indirectly, not as so - Dkron fixes this problem by, as they claim, being the only existing scheduler with no SPOF.

Therefore, with acceptable performance, whilst providing Reliability, Dkron is an interesting solution for use-cases that must guarantee fault-tolerance.

3 Benchmarking

Due to the project time constraints, an extensive literature review was not performed, however a great portion of this project's approach, execution and reporting is done based on the acquired knowledge in the course.

During the limited research, I could not find a convincing standard benchmark to use or be based on. Instead, I focused on researching the particularities of Dkron and exploiting those to gather conclusions and, admittedly, confirm some deductions I was forming throughout Dkron's design analysis.

One of the clearest characteristic for performance analysis in Dkron is its use of Raft to guarantee reliability. This, along the fact that all the dkron cluster members of kind Server persist, between themselves, all the job responses' logs.

Thus, the intuitive conclusion for Dkron is: **1)** Increase Dkron Servers to improve fault-tolerance, but increases overhead, **2)**Increase Dkron Agents to improve performance.

Nevertheless, solely increasing Agents, even with unlimited costs, is not an infinite answer. Even more so considering that, unless the https://dkron.io/docs/usage/target-nodes-spec, every member in the cluster will run the job.

With very limited, information on Dkron's scaling capabilities, one can assume, contention and crosstalk will increase as most systems do. Without a formal baseline to which to compare with, I took the liberty to thinker with a synthetic workload to push the limits of Dkron whilst considering the University Scalability Law and its properties.

$$X(N) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)}$$

where:

- $X(N)$ is the throughput of the system with N resources.
- N is the number of resources (e.g., processors, nodes).
- α is the contention coefficient, representing the contention delay due to resource sharing.
- β is the coherency coefficient, representing the delay due to the need to maintain coherence across resources.

Having this, considering the Universal Scalability Law, the system characteristics that would most affect Dkron scalability could be:

Increase in Dkron Servers may increase contention, since there are serial parts of the dkron process like, receiving, dispatching, and storing a job. The serial part of the job itself, will depend on the process to execute associated to the job, this can be anything. Regarding crosstalk, this can be assumed to be a big part of these kinds of members since a server needs frequent communication between its consensus peers, e.g., for leader election, and state coherence.

Increase in Dkron Agents may increase contention, having more members in the cluster increases the cluster provisioning, maintenance, and load balancing overhead. Regarding crosstalk, although not as much as Dkron Servers, but Agents need to join the cluster and become known by the Servers for job dispatch, as well every new agent is a new running job that needs (by default) to be stored by servers, requiring state coherence.

Built Heuristic Benchmak Dkron does not make any claims on its performance and scalability capacities. Joined to the fact that I could not find any relevant ready-made or standard benchmarks, heuristics were used.

The benchmark was built with K6 in JavaScript. I did experiment with the following: CPU-focused job, doing a high bound loop with arithmetic computations; and Memory-focused job, read data and pipe it. However, these jobs proved to be very heavy for the Dkron pods, as it would cause crashes and even no fault recovery unless manually fixed.

The final resulting benchmark used to extrapolate the metrics, does 2 simple jobs that execute the same, an echo, but in different manners: 1) schedule echo, this job will run indefinitely every time, associated to the frequency defined, e.g., (echo) every 2 seconds. 2) immediately request echo, this job is a basically a post request that expects an immediate response. These jobs are either ran by the pod that received the job, or dispatched, by a Server, to another pod.

Regarding immediately requesting echo, k6 provides an interface to simulate multiple virtual users making requests.

```
export const options = {
    stages: [
        { duration: "60s", target: 20 },
        { duration: "60s", target: 60 },
        { duration: "60s", target: 0 },
    ]
};
```

As observed in the code in FIGX, this benchmarks runs for 3 minutes ramping up virtual users to 20 during the first sixty seconds, then ramping up from 20 to 60 in the next sixty seconds, and then ramping down.

This benchmark has the goal of extremely stressing the cluster with an increasing amount of workload, i.e., scheduled jobs. A schedules job once created, will run indefinitely for its frequency unless deleted, meaning that as time passes, the stress increases.

The script can be analysed in */benchmark/benchmark.js*. The purpose of this was to be able benchmark the response time over an environment where Dkron is under a big stress. With this, we could extrapolate from k6 the response time and error rate, which was the main goal. The reasoning behind this is that a scheduler is a system that, under well-organized definitions, is set in such a way where it should have always the opportunity to run the jobs it is instructed to. Having this, my way of thought for this benchmark was to see how well Dkron handles a peak amount of stress (3 minutes benchmark) and can handle state coherency, state persistance, and job dispatch and execution.

Other performance metrics like throughput and utilization could also be explored. However, response time, which includes latency seemed more suitable for a job scheduler. More discussion on this topic of metrics can be found in 4

3.1 Dkron Bottlenecks

1. Decompose a request into a pipeline of stages.

A default request (recurrent + concurrent) can be sectioned into the following: **1)** Request done to AKS Load Balancer, **2)**Load Balancer forwards to one of the pods (servers or agents), **3)** If the receiving pod is not the leader, sends an event to the leader warning about the request, **4)** The leader starts a Raft Consensus to reach a common value with the quorum and schedule the job, **5)** If all goes well, the job is scheduled, **6)** A scheduled job is queried by a Server and either executed or dispatched to an Agent, **7)** The pod executing the job, executes it and responds with the execution result, **8)** The execution result is stored in the Server's BuntDB instance.

Not considering the Raft consensus algorithm as I believe analysing it is out of scope of the project, one of the stages that could be improved are stage 6, 7, and 8. Since Dkron seems to already use concurrency, I will skip that technique for hiding latency.

The exploit of workload properties could be used: by having optimized nodes of Dkron for certain kinds of operations, the use of Dkron's Tag-based job dispatch could be used in communion with these optimized nodes. Therefore, if the Dkron workloads prove to be heterogenous, this could be a good solution to explore.

Similarly, if the Dkron is used for chained jobs, the Dkron admin could look into any request patterns and look for optimizations in chaining.

Queuing could also make a positive impact. As I did on my benchmark, Dkron may have periods of peak stress, this creates instability on the nodes and leads into a snowball effect of failed requests and increased response time. By having a queue to accommodate short overload bursts when load is greater than capacity, Dkron could improve stability.

Batching could be used: Instead of sending one request at a time, the server nodes could group requests for a batch and send, e.g., to a Dkron Agent, where the Agent would concurrently execute them and respond with the state logs in one go also. This also opens opportunities to drop some possibly unnecessary requests or reorder requests.

3.2 Experimental Design

After analysing Dkron's design and particularities, the Experimental Design was initiated with the goal to extrapolate the most possible information out of the decisions that will be discussed below.

A Fractional Factorial Design was used in order to use more factors (minimum of 6), whilst reducing the amount of experiments required to be done.

A 2^{k-p} was used with k=*6* and p=*3* combined factors, resulting in a total of 8 experiments.

Factors As can be observed, the factors are the following: HasOnDemandJobs, CPU Limit (m), Memory Limit (GiB), Dkron Servers, Dkron Agents, Concurrent. Regarding the rationale for the choice of each factor. When deciding for factors, after analysing the particularities of Dkron, some questions were raised:

1. How can Dkron handle high peak overload and still function, i.e., do Raft, execute and dispatch?
2. How affected are immediate requests response time when Dkron is on high load?

The conclusions from the analysis and consequently, the attempt to answer the above questions can be found in subsection 4

HasOnDemandJobs: Dkron is mainly built to schedule recurring jobs, and this github issue confirms that it was not initially planned to support immediately, one time, executed jobs. The main exploratory idea was to analyse how immediate jobs are reacted to i.e. if they have some threshold to be accepted in case all the pods are busy, or if in the case all the pods are busy, the request is simply rejected. **CPU Limit(m):** Dkron uses Raft to achieve consensus between the Dkron Agents. Distributed consensus protocols usually are expected to be CPU and Memory bound therefore, this factor is relevant to observe the impact of the Dkron Servers realizing consensus and providing reliability. **Memory Limit(GiB):** Likewise, due to the actions of state management and replication being heavily required and frequent in these kinds of protocols, Memory also seemed a relevant factor to take into account. **Dkron Agents:** A Dkron Agent has a very simple goal, to execute jobs dispatched by Dkron Servers. Having this simple pod that does not participate on consensus and only executes jobs seemed to be the clear factor to increase for performance gains. **Dkron Servers:** Dkron Servers do what Dkron Agents do plus scheduling of jobs, handling data storage and participating on leader election. Since these 2 present very distinguishable characteristics, it seemed relevant to analyse their interaction. **Concurrent:** Dkron is states that if no Dkron Agent/Server is available

Factor Levels Acknowledging there is the method of 2^k Factorial Design, it would be interesting to try with some minimum and maximum values to look for *unidirectional effects* and in case the factor is has a unidirectional effect decide on the values from there. However, from deploying an Azure Kubernetes Service to Dkron becoming ready was taking approximately 10 minutes and doing a Rolling Update was proving to be challenging, from the non-clear dkron's helm chart to the particular mechanism of Dkron's clustering (**??**).

Considering those challenges and the limited time, the approach to find the levels was heuristics: For computational resources, i.e., CPU Limit and Memory Limit, an optimal point between these resources and prices was the consideration. For the ratio between Dkron Agents and Dkron Servers, an initial goal was set, until challenges were found, making it unable to use the desired levels, servers (3 to 5) and agents (0 to 3). Below an oversimplification of the challenge will be shared. For the benchmark workload parameters, these are simply binary values.

As for the challenge with setting the desired Dkron Agents replicas, it seems dkron helm chart is still neither stable, nor prioritized by Dkron's team, an unfortunate late discovery as it was unfeasible to leave the already too invested kubernetes and helm deployment leaving. the only option to analyse with Dkron Agent as more of a binary value, 0 or 1. Particularly, the issues seems to be with one of the templates *agent-deployment.yaml* or *agent-hpa* which is ignoring the agent's set replica count and only creates 1, independently of the set value, 0 or 100.

Confounded Factors Confounding is used to reduce the number of ran experiments while being able to use a broader range of factors. It is relevant to note that reducing the number of experiments in favor of time and cost savings and sole increase in factors is not the best approach. Instead, it should be given preference to confound factors while maintaining a higher level of experiments where their experiment error can be considered in the effect results.

When confounding factors we should confound 1) factors where the interaction is probable to be negligible, 2) factors where combined effects are not so relevant to the analyis.

- **HasOnDemandJobs + CPU Limit for Dkron Servers:** these 2 confounded factors are unlikely to interact since the expected jobs done by a scheduler are not heavy, and sometimes of workflow continuation or triggering. Both of these confounded factors would interact more if confounded with, e.g., Dkron Servers, for reasons of consensus overhead already explored.
- **HasOnDemandJobs and Memory confounded for Dkron Agents:** same rationale as HasOnDemandJobs + CPU Limit.
- **CPU Limit and Memory Limit for Concurrency**: same rationale as HasOnDemandJobs + CPU Limit.

4 Results

In this section I provide the Factorial Analysis results.

Figure 7 provides the domains of the factors under analysis and that were used to build the benchmarks. Figure 8 provides the sign table with the original values of each factor, allowing for a complete overview of each benchmark setup. Figure 9 represents the same table of Figure 9, but with the respective signs of each factor value/category. These Figures can be consulted in the Annex.

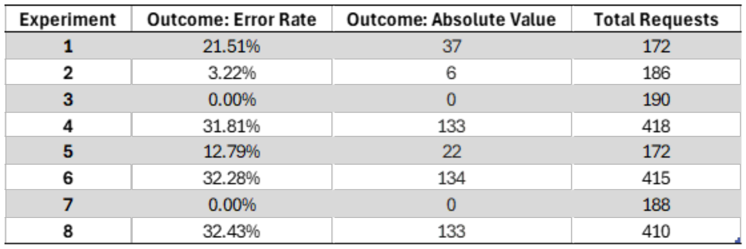
Under this experimental design, the goal was to evaluate the failed requests of each benchmark. To quantify the effect of each factor on failed requests, the sign matrix was tested for orthogonality. Considering the matrix is composed by 8 experiments and 7 factors, it comes that the matrix is orthogonal under the following conditions:

– Sum of each column is zero.
– Sum of products of each two columns is zero.
– Sum of column squares is $2^{7-4} = 8$.

The matrix (consult Figure 9 in Annex) satisfies all conditions of orthogonality. Hence, the calculation of each effect comes straightforward: $q_k = \sum_{k=1}^{K} \sum_{p}^{P} s_{pk} \times y_p$, where q_k is the effect of factor k, s_{pk} is the sign of factor k in experiment p, K is the total number of factors under analysis, and P is the total number of benchmarks, or experiments, conducted.

A remark must be made regarding the outcome being evaluated. The failed requests of each benchmark can be analysed in rate or in absolute value. If the primary goal is to understand relative performance, then one should analyse the percentage of failed requests. This is especially useful if benchmarks have varying total numbers of requests, which is the case, and one aims to compare each experiment on a normalised scale. On the other hand, if the primary goal is to understand the impact in terms of volume, one should opt for the absolute number of failed requests. This is more relevant to evaluate the overall impact or size of failures. The following paragraphs provide a brief summary of the factorial analysis conducted for both scenarios, once I believe they are both important.

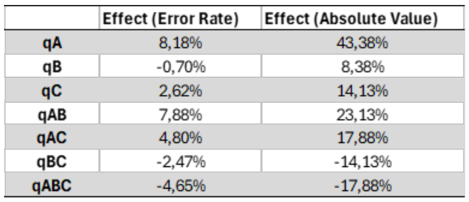
The outcome expressed in rate and absolute value, as well as the total number of request per benchmark, are provided in Figure 1.



Experiment	Outcome: Error Rate	Outcome: Absolute Value	Total Requests
1	21.51%	37	172
2	3.22%	6	186
3	0.00%	0	190
4	31.81%	133	418
5	12.79%	22	172
6	32.28%	134	415
7	0.00%	0	188
8	32.43%	133	410

Fig. 1. Experiment Results and Total Number of Requests per Experiment.

The factorial analysis results for each factor change, depending on the outcome being considered. Please consult the results in Figure 2.



	Effect (Error Rate)	Effect (Absolute Value)
qA	8,18%	43,38%
qB	-0,70%	8,38%
qC	2,62%	14,13%
qAB	7,88%	23,13%
qAC	4,80%	17,88%
qBC	-2,47%	-14,13%
qABC	-4,65%	-17,88%

Fig. 2. Effects of each Factor, considering the different outcomes.

It is possible to assess that the effect of the factors changes in magnitude, depending on the outcome being considered, and factor B also changes in sign (or its in impact). Independently of the outcome being considered, factor A (Has On Demand Jobs) is the one that contributes more towards requests failure. This means that a higher value of Has On Demand Jobs is likely to lead to a higher failure number (or rate) of requests. Other factors that contribute to the failure of requests are, in descending order, interaction AB (DKron Servers), interaction AC (DKron Agents) and factor C (Memory). On the other hand, interaction BC (Concurrent) helps mitigating the number or rate of failed request, as well as the interaction ABC. Factor B is negligible when the outcome is expressed in rate.

However, a small note on utilization, although not formally registered in the report, I was observing the pods CPU and Memory utilization.

5 Conclusion

Dkron shows to provide reliability and have two distinct kinds of nodes Servers and Agents. Both seem to play a crucial role, Servers to guarantee reliability and Agents to provide scaled performance.

Nevertheless, Dkron's helm chart is still unstable. Dkron Agents in particular was a factor that unfortunately could not be configured in the desired way to extrapolate clear conclusions and answer the initial questions.

5.1 Limitations

Dkron's helm chart is still not stable for an easy deployment. Documentation is lacking, and it seems there is no stable helm chart for dkron. Additionally, during bug fixing and deployment, I was left with the idea that [a lot more other developers face their own unexpected issues with dkron's helm chart](https://github.com/distribworks/dkron/issues?q=helm).

6 Annexes

6.1 Metrics Visualisation

The following Figures provide some visualisations about each experiment. All plots have the same x-axis order: NOs3a0-250-1NO, NOs3a0-500-2YES, NOs3a1-500-1NO, YESs3a1-250-2NO, NOs7a1-250-1YES, YESs7a1-500-2YES, YESs7a0-500-1NO, NOs7a0-250-2NO. This corresponds to the following experiment order: 2, 7, 3, 6, 1, 8, 4, 5.

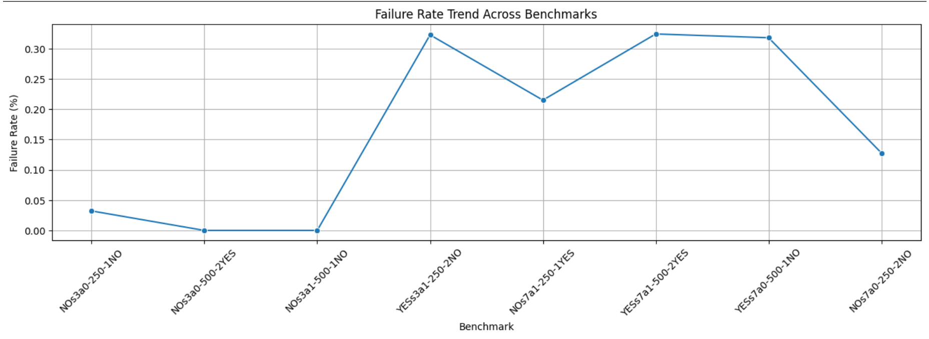


Fig. 3. Failure Rate Trend Across Benchmarks

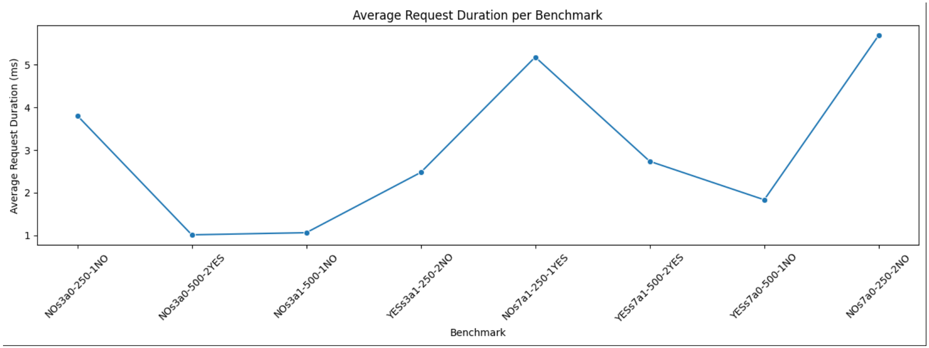


Fig. 4. Average Request Duration Per Benchmark

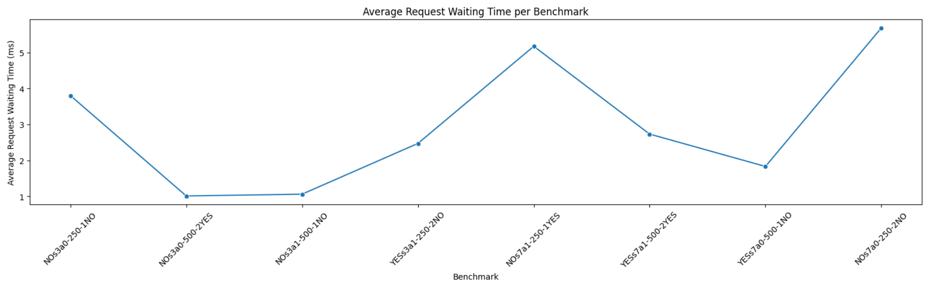


Fig. 5. Average Request Waiting Time Per Benchmark

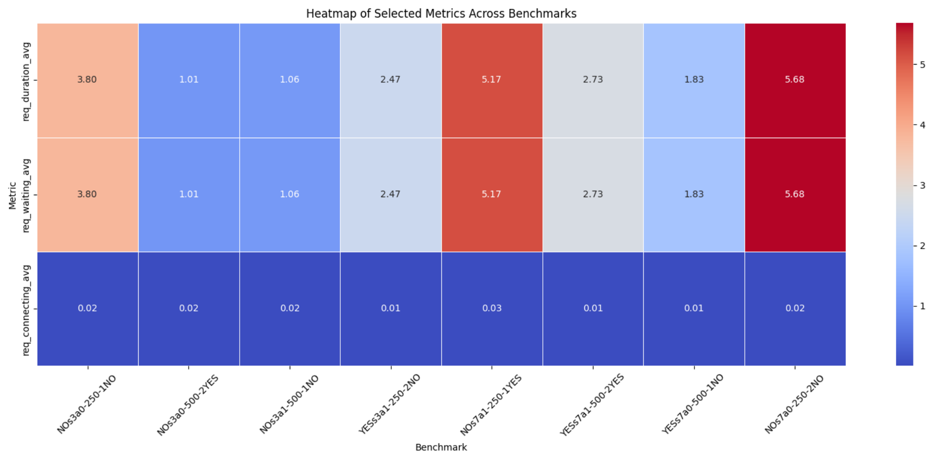
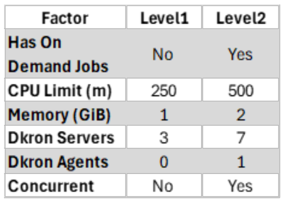


Fig. 6. HeatMap of selected metrics

6.2 Factorial Analysis

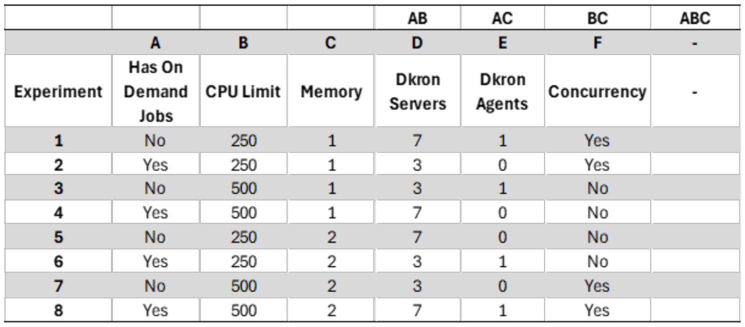
Figure 7 provides the domains of the factors under analysis and that were used to build the benchmarks.



Factor	Level1	Level2
Has On Demand Jobs	No	Yes
CPU Limit (m)	250	500
Memory (GiB)	1	2
Dkron Servers	3	7
Dkron Agents	0	1
Concurrent	No	Yes

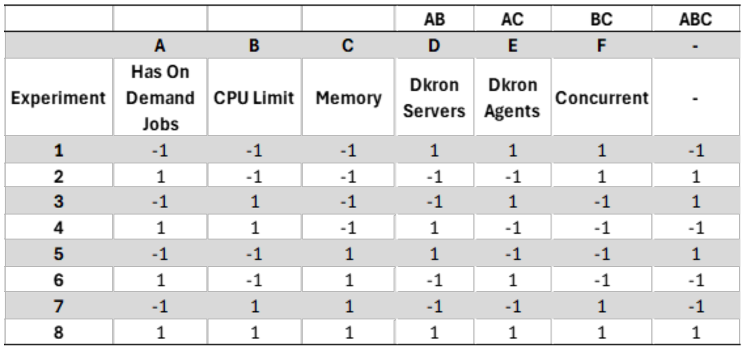
Fig. 7. Domain of the Factors.

Figure 8 provides the sign table with the original values of each factor, allowing for a complete overview of each benchmark setup.



				AB	AC	BC	ABC
	A	B	C	D	E	F	-
Experiment	Has On Demand Jobs	CPU Limit	Memory	Dkron Servers	Dkron Agents	Concurrency	-
1	No	250	1	7	1	Yes	
2	Yes	250	1	3	0	Yes	
3	No	500	1	3	1	No	
4	Yes	500	1	7	0	No	
5	No	250	2	7	0	No	
6	Yes	250	2	3	1	No	
7	No	500	2	3	0	Yes	
8	Yes	500	2	7	1	Yes	

Fig. 8. Sign Table with Values.



				AB	AC	BC	ABC
	A	B	C	D	E	F	-
Experiment	Has On Demand Jobs	CPU Limit	Memory	Dkron Servers	Dkron Agents	Concurrent	-
1	-1	-1	-1	1	1	1	-1
2	1	-1	-1	-1	-1	1	1
3	-1	1	-1	-1	1	-1	1
4	1	1	-1	1	-1	-1	-1
5	-1	-1	1	1	-1	-1	1
6	1	-1	1	-1	1	-1	-1
7	-1	1	1	-1	-1	1	-1
8	1	1	1	1	1	1	1

Fig. 9. Sign Table.

References

1. Author, F.: Article title. Journal **2**(5), 99–110 (2016)
2. Author, F., Author, S.: Title of a proceedings paper. In: Editor, F., Editor, S. (eds.) CONFERENCE 2016, LNCS, vol. 9999, pp. 1–13. Springer, Heidelberg (2016). `https://doi.org/10.10007/1234567890`
3. Author, F., Author, S., Author, T.: Book title. 2nd edn. Publisher, Location (1999)
4. Author, A.-B.: Contribution title. In: 9th International Proceedings on Proceedings, pp. 1–2. Publisher, Location (2010)
5. LNCS Homepage, `http://www.springer.com/lncs`, last accessed 2023/10/25