**Vraj Ketankumar Shah Nithik Pandya Ayush Dodia**

**Vs921 nhp46 and179**

**Distributed Billing Aggregation: A MapReduce Approach Using Apache Spark on Kubernetes**

Github Repo: [repo-link](https://github.com/vraj2131/Distributed-Billing-Aggregation-A-MapReduce-Approach-)

Demo Video:

1. **Introduction: -**

In any application where users interact through APIs, tracking and billing based on usage is critical for transparency and cost control. Every API request generates a log line with information like which user made the request, what action they took, and how long it took. Aggregating these logs into total usage and cost per user may sound simple, but as the volume of data grows, from thousands to millions of records, processing it efficiently becomes a real challenge. Initially, we used a straightforward Python script to read and process these logs, but this approach couldn’t scale and lacked fault tolerance. Aggregating such logs to compute usage-based billing per user is straightforward at a small scale, but becomes challenging as the volume grows. A naive script that reads **10,000** log lines or more sequentially can struggle or take too long when the data scales to millions of entries. Moreover, running on a single machine introduces a single point of failure and limited throughput.

To solve this, we turned to Apache Spark and Kubernetes. Spark allows us to distribute the billing computation across multiple cores or machines, using a MapReduce-like pattern to process logs in parallel. Kubernetes gives us the ability to dynamically scale resources, isolate workloads, and deploy repeatably in both local and cloud environments. Through this project, we implemented and compared four different modes: a naive local script (single-threaded, on one host), a Spark standalone cluster using Docker Compose (with its MapReduce-style parallel processing), a Spark-on-Kubernetes setup using Kind (by using multiple executors in parallel), and a fully cloud-native deployment using AWS EKS. Each mode builds on the last, improving scalability, speed, and operational reliability.

In summary, the goal is to evolve from a basic script to a robust distributed solution. We implemented and evaluated four approaches in increasing order of complexity and scalability:

* Naive single-process script,
* Local Spark standalone cluster,
* Kubernetes-based Spark cluster on Kind (Docker), and
* Spark cluster on AWS EKS.

1. **Input: -**

We have 10000 rows of API call. Every API call in our platform produces a log entry with details like timestamp, user ID, action, and duration. For example, a login event might appear as:

2025-05-02T23:16:50Z user115 login 200 1568ms

* The first part is a timestamp.
* Second is the user ID
* Next is the type of API call
* Fourth is the status code of the API call
* Last is Duration

1. **Methodology**

We will demonstrate the approach we used for implementing all 4 methods to compute the total billing cost for each user.

Commands to execute before starting to work:

* Activate virtual environment:

source .venv/bin/activate

* Install requirements:

pip install -r requirements.txt

* Naive Implementation: -

The naive implementation is a plain Python script running on a single machine. It does not use any special framework – just file I/O and Python dictionaries in memory. This script reads the entire log file line by line and computes totals in a single thread.

The script opens the log file and iterates through each line. For each log entry, it extracts the user, task (action), and duration. It then looks up a predefined rate for the task (from environment variables like RATE\_login, RATE\_createOrder, etc.) and calculates the cost for that entry (cost = duration\_ms \* rate). These per-line results are accumulated in a dictionary keyed by user.

This logic is implemented in a straightforward loop:

totals = {}

for line in log\_file:

user, task, duration = parse\_line(line)

cost = duration \* rates.get(task, 0.0)

if user not in totals:

totals[user] = {'total\_duration\_ms': 0, 'total\_cost': 0.0}

totals[user]['total\_duration\_ms'] += duration

totals[user]['total\_cost'] += cost

Each user's total\_duration\_ms and total\_cost are incremented for every line belonging to that user. The rates for each API operation are supplied via environment variables.

This approach is the simplest but also the least scalable. It uses a single process and one CPU core. For our sample of 10,000 log lines, it completed quickly since the data size is modest. The script utilized essentially 100% of one CPU while parsing the file, but it cannot leverage multiple cores or machines. There is no parallelism – the logs are processed one by one. If the dataset were 100× larger, the processing time would grow roughly 100× as well, since there's no distribution of work. Moreover, there’s no built-in fault tolerance; if the script crashes midway, we have to restart from scratch. These limitations motivated moving to a Spark-based solution for better performance and resilience.

Command to execute it: -

python3 src/mapreduce\_billing/naive\_aggregation.py \

--input-path data/api\_logs.txt \

--output-path data/results/billing.txt

This command reads data/api\_logs.txt (our collected API log sample) and writes the aggregation results to data/results/billing.txt.

* Local Standalone Cluster: -

In this mode, we set up an Apache Spark cluster on a single machine using Docker containers. The cluster followed Spark’s standalone deployment mode: one Spark master and one Spark worker running in separate containers, all orchestrated with Docker Compose. We packaged our application code into a Docker image, so the same image can serve as master, worker, and submission client. Using Docker Compose, we defined services for a Spark master and a Spark worker container. The master container ran start-master.sh (listening on port 7077 for Spark jobs), and the worker container ran start-worker.sh pointing at the master’s URL. In our compose file, the worker was configured with SPARK\_WORKER\_CORES=2 and SPARK\_WORKER\_MEMORY=4g, meaning it could run up to 2 CPU cores worth of tasks. The master UI was available on port 8080 for monitoring. Both containers shared a volume for the logs data (./data on host mounted to /app/data in containers) so that the worker could read the api\_logs.txt file.

Once the cluster was up, we ran our aggregation job on it. We did this by running a third container that uses the same image to execute the Spark submit script. This invoked our submit\_spark\_job.sh script inside a container, which in turn called spark-submit with the appropriate configuration. The Spark job used our MapReduce functions to perform the aggregation. Under the hood, it created a Spark RDD from the text file and applied a map and reduce.

The billing aggregation logic is divided into two main phases: the map process and the reduce process.

In the **map process**, each log line is read and broken down to extract the user ID, the type of API task performed, and how long that task took. A pre-defined rate is associated with each task type, and this rate is multiplied by the duration to calculate the cost of that individual API call. The output of this phase is a set of key-value pairs where the key is the user, and the value is a tuple containing the duration and the calculated cost.

In the **reduce process**, all entries with the same user key are grouped together. The durations and costs across all of that user's records are added up to produce a final total. This results in a compact summary for each user, showing how long they used the system and how much their usage cost in total. This two-step process allows the system to efficiently process and aggregate large volumes of API logs in parallel.

If our data were larger, the Spark approach would start to outperform the naive script by utilizing multiple cores. Still, with only one node, the speed-up is limited. Another point is resilience: if the worker failed, the job would fail since we had no alternate workers.

A diagram of a work flow

Description automatically generated

Command to execute it: -

First Step is to make sure ENVIRONMENT= local in 3 files:

.env (line 3), config.py (line 15), spark\_job.py (line 38)

Then next go in submit\_spark\_job.sh and check **line 125:**

remove local:// if it is present. That line should look like:

/app/src/mapreduce\_billing/spark\_job.py \

Now in the terminal where venv is activated and requirements are installed, start executing it one after the other.

docker build -t distributed-billing-spark:local .

chmod +x scripts/submit\_spark\_job.sh

docker-compose up -d spark-master spark-worker history-server

docker-compose run --rm spark-submit

docker-compose down

History Server can be accessed at: <http://localhost:18080/>

Spark UI can be accessed at: <http://localhost:8080/>

The last command will shut down the container.

| **Component** | **Configuration** |
| --- | --- |
| **Spark Version** | 3.4.1 |
| **Deployment** | Standalone mode |
| **Worker Count** | 1 |
| **Worker Memory** | 4 GB |
| **Executor Cores** | 2 |
| **Executor Instances** | 3 |
|  |  |
|  |  |

* Spark on Kubernetes (using Local Kind Cluster)

The third mode containerized the Spark job within a Kubernetes cluster using **Kind** (Kubernetes-in-Docker) to simulate a cloud environment on a local machine. In this setup, Spark runs in Kubernetes cluster mode: the Spark driver and executors run as pods managed by the Kubernetes control plane. This approach is fully cloud-native – Spark leverages Kubernetes for deployment, scaling, and isolation of compute pods. We chose Kind to emulate a real K8s environment for testing. We also configured Kubernetes resources needed by Spark: for example, we ensured there was a service account (spark-serviceaccount) with the necessary permissions for the Spark driver to create executor pods. Spark was configured to run in cluster mode, with the driver and executor pods managed by Kubernetes itself. We also enabled dynamic allocation, allowing Spark to scale executor pods automatically within defined limits. The job used a hostPath-mounted volume so that log input files and result outputs could be shared between the host system and the pods.

Initially, the Spark job ran with only two data partitions, which resulted in underutilization of resources—only two out of four executor pods were active, while the others remained idle. After adjusting the number of partitions to four, all executors were effectively used, with one task assigned per pod. This significantly improved performance and showcased the importance of proper partitioning to match the level of parallelism available. The Kind mode provided a full end-to-end Kubernetes workflow, including resource management, pod scheduling, and log collection, giving us a close-to-production experience while still running entirely on a local machine.

Another difference in this mode was the **isolation and resilience** Kubernetes provides. Each Spark executor ran in its own pod, and if one had failed, Kubernetes could reschedule it (Spark would retry the task on a new pod). Our local Kind cluster had limited resources, but the setup mimics how on a real multi-node cluster, executors could be spread across nodes. We also took advantage of Kubernetes logging and monitoring to see the progress of the job. In essence, the Kubernetes cluster (even though it was a single-node Kind cluster) was able to leverage all available parallelism. The driver coordinated 4 parallel tasks in the map phase and then 4 tasks in the reduce phase (each executor likely handled a portion of the reduced data as well).

We created several yaml files to configure how the cluster is created, what permissions it will have, how many workers provisioned, etc. We also included History server where we can check different logs after a process is successfully executed.

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

Here we can see how it takes the text file, how map and reduce functions will work, etc.

Commands to execute: -

First Step is to make sure ENVIRONMENT= kub in 3 files:

.env (line 3), config.py (line 15), spark\_job.py (line 38)

Then next go in submit\_spark\_job.sh and check **line 125:**

add local:// if it is not present. That line should look like:

local:///app/src/mapreduce\_billing/spark\_job.py \

Then one after the other start executing it in the terminal:

docker build -t distributed-billing-spark:latest .

kind create cluster --name spark-cluster \

--config config/k8s/kind-spark-cluster.yaml

kind load docker-image distributed-billing-spark:latest --name spark-cluster

kubectl config use-context kind-spark-cluster

kubectl get nodes

kubectl delete clusterrolebinding spark-sa-admin 2>/dev/null || true

kubectl create clusterrolebinding spark-sa-admin \

--clusterrole=admin \

--serviceaccount=default:spark-serviceaccount

kubectl delete configmap app-env

kubectl create configmap app-env --from-env-file=.env

kubectl apply -f config/k8s/spark-serviceaccount.yaml

kubectl apply -f config/k8s/spark-master-ui-svc.yaml

kubectl apply -f config/k8s/history-server.yaml

kubectl apply -f config/k8s/fluent-bit-config.yaml

kubectl apply -f config/k8s/fluent-bit-daemonset.yaml

kubectl apply -f config/k8s/spark-history-ui-svc.yaml

Now in new terminal:

kubectl config use-context kind-spark-cluster

kubectl port-forward svc/spark-history-ui 18080:18080

Here, history server is started which could be accessed at: <http://localhost:18080/>

Now back in the original terminal:

kubectl apply -f config/k8s/billing-job.yaml

To get logs in terminal for cluster:

POD=$(kubectl get pods -l job-name=billing-on-demand \

-o jsonpath='{.items[0].metadata.name}')

kubectl logs $POD -c spark-submit

For logs of Billing aggregation job:

kubectl logs billing-aggregation-$( \

kubectl get pods \

-l spark-app-name=billing-aggregation,spark-role=driver \

-o jsonpath='{.items[0].metadata.name}' | cut -d'-' -f3 \

)-driver \

-c spark-kubernetes-driver

To watch the jobs:

kubectl get pods –watch

To delete everything:

kubectl delete job billing-on-demand

kubectl delete -f config/k8s/history-server.yaml

kubectl delete -f config/k8s/spark-master-ui-svc.yaml

kubectl delete -f config/k8s/spark-serviceaccount.yaml

kubectl delete -f config/k8s/spark-history-ui-svc.yaml

kubectl delete configmap app-env

kind delete cluster --name spark-cluster

| **Component** | **Configuration** |
| --- | --- |
| **Cluster Tool** | Kind |
| **Spark Mode** | Kubernetes cluster mode |
| **Driver Pod** | Runs in cluster; created by spark-submit |
| **Executor Pods** | Dynamically allocated (min: 2, max: 10) |
| **Partition Count** | 4 |
| **Worker Count** | 5 |
| **Worker Memory** | 4 GB |

* AWS EKS: -

The final mode ran the Spark aggregation on a **managed Kubernetes cluster (AWS EKS)**. This closely mirrors a production deployment. The Spark driver and executors run as pods on a multi-node EKS cluster, and the input data is stored in AWS S3 rather than on a local disk. We still use Spark’s Kubernetes cluster mode with dynamic allocation, but now we have the cloud’s scalability at our disposal.

We used AWS console to create EKS cluster. For that we also created 2 IAM roles one for EKS cluster and another for Worker Node policy to manage.

We configured AWS in our terminal, logged in, exported environment variables.

Like previous steps we created a docker image, which we tagged into ECR and pushed it.

On EKS, the Spark job ran with **dynamic allocation** enabled (just like in the Kind case). The difference here is that the EKS cluster had multiple nodes and more CPU capacity. Spark’s configuration was set to start with 4 executors and scale between 2 and 10 as needed. The input data was fetched directly from S3 via the S3A connector, meaning each executor pod could pull its portion of the log file from the cloud storage in parallel. The MapReduce logic didn’t change – it still created an RDD and performed the aggregation.

The job completion time on EKS was roughly on par with the improved Kind run – any additional speedup was minor given the data size (the overhead of container launch dominates for small jobs). However, the **capacity to scale** is much greater. If we had a much larger log file (say, 100 million lines on S3), Spark on EKS could have allocated more executors (up to our max of 10, or we could configure higher) and even requested more cluster nodes if configured with cluster autoscaling. This means our solution can grow to handle production volumes by simply adjusting configuration, without code changes.

Another difference is **integration with cloud services**. By using S3 for input, we didn’t need to manually move data onto the processing nodes – Spark handled it. This decouples storage from compute and is aligned with cloud best practices. We also benefited from AWS-managed infrastructure: the Kubernetes control plane is managed by AWS (improving reliability), and we could leverage IAM roles or secrets management for secure access to data.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

The 2 snapshots are of when I created the cluster and worker nodes on EKS. I have deleted the cluster as well as all instances of worker nodes. I was new to using AWS and was skeptical about the cost and how it would be calculated so it won’t be possible to execute in your system.

Commands which I used:

First Step is to make sure ENVIRONMENT= aws in 3 files:

.env (line 3), config.py (line 15), spark\_job.py (line 38)

Then next go in submit\_spark\_job.sh and check **line 125:**

add local:// if it is not present. That line should look like:

local:///app/src/mapreduce\_billing/spark\_job.py \

Then in terminal:

aws configure

AWS Access Key ID [None]:

AWS Secret Access Key [None]:

Default region name [None]: us-east-2

Default output format [None]:

export EKS\_CLUSTER\_NAME=billing-aggregation-eks

export AWS\_REGION=us-east-2

export AWS\_ACCOUNT\_ID=$(aws sts get-caller-identity \

--query Account --output text)

aws eks update-kubeconfig \

--region $AWS\_REGION \

--name $EKS\_CLUSTER\_NAME

aws ecr create-repository \

--repository-name distributed-billing-spark \

--region $AWS\_REGION

aws ecr get-login-password \

--region $AWS\_REGION \

| docker login \

--username AWS \

--password-stdin \

$AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com

docker build -t distributed-billing-spark:latest .

docker tag distributed-billing-spark:latest \

$AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/distributed-billing-spark:latest

docker push \

$AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/distributed-billing-spark:latest

SPARK\_K8S\_IMAGE=$AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/distributed-billing-spark:latest

kubectl config current-context

kubectl get nodes

kubectl create namespace billing

kubectl config set-context --current --namespace=billing

kubectl create configmap app-env --from-env-file=.env

kubectl apply -f configs/k8s/billing-job-aws.yaml

kubectl logs -n billing -f job/billing-on-demand

kubectl delete job billing-on-demand -n billing

| **Component** | **Configuration** |
| --- | --- |
| **Cloud Platform** | AWS EKS (Elastic Kubernetes Service) |
| **Cluster Nodes** | Managed Node Group with t2.medium EC2 instances |
| **Node Specs** | 2 vCPUs, 4 GiB RAM per node |
| **Disk Size per Node** | 20 GiB EBS volume |
| **Spark Deployment Mode** | Kubernetes cluster mode |
| **Container Image** | Stored in AWS ECR |
| **Dynamic Allocation** | Enabled – Min: 2 executors, Initial: 4, Max: 10 executors |

1. **Results**

We measured and observed the performance of each mode, not only focusing on execution time, but also on how well the executors were utilized and how tasks were distributed across available resources.

* The naive Python script completed the 10k-line aggregation in only a couple of seconds on a single machine. It used a single CPU core at near 100% during execution. There was **no parallelism** – one record was processed at a time – which means the CPU utilization was limited to one core even though the machine had multiple cores available. While this was very fast for a small input, it doesn’t scale well. In real world scenarios data could be in terabytes where this approach will fail terribly.
* The Spark standalone mode introduced parallelism, but its benefits were limited by the single-node cluster. The total execution time was slightly longer than the naive script for 10k lines, mainly due to Spark’s startup and coordination overhead. However, because we only had one worker with 2 CPU cores, the tasks had to be executed in waves. The Spark master launched up to 2 tasks concurrently (utilizing the 2 cores), then the next 2 tasks after the first ones finished. All tasks were executed by a single executor process on the single worker node (since effectively only one executor could be scheduled given the worker’s resources). This meant **executor utilization** was not high in quantity (only one executor process), but that executor was kept busy.
* In the first Kubernetes run, we encountered a scenario where the job was not fully optimized. Spark defaulted to 2 partitions for our input data, which resulted in **2 tasks** for the main computation. According to our dynamic allocation settings, Spark started 4 executor pods on the Kind cluster (because we allowed up to 4 and it eagerly started the initial executors). However, only 2 of those executors actually received tasks to execute. Each active executor processed roughly half of the log lines. The other 2 executor pods remained idle (they started up, registered with the driver, but had no partitions assigned to them). The job still finished correctly, but the **executor utilization** was only 50% – half the executors did not work.
* For the second run on Kubernetes, we improved the parallelism by increasing the number of partitions to 4. This small change had a significant impact. The Spark job now created **4 tasks** for the map stage, which matched the number of executors that the cluster was configured to use. Consequently, all 4 executor pods were utilized. The driver distributed one task to each executor. Each executor pod processed about 2,500 log lines and computed partial sums for its subset of users. With all executors working in parallel, the job’s **wall-clock time decreased** (nearly half of the 2-partition run, as expected for double the parallelism, minus some overhead). The executor utilization was effectively 100% during the processing: every executor was crunching data. In the subsequent reduce stage, work was also spread across the 4 executors (each handling a portion of the key space to combine the results). This run demonstrated the benefit of aligning the number of partitions with the degree of parallelism available. The overall execution was more efficient, and the Kubernetes resources were used optimally.
* The EKS run was our most robust test, leveraging a real multi-node cluster. With 4 partitions (as in the improved run above), Spark again launched 4 executor pods. In a typical EKS setup with multiple nodes, these executors could be distributed across different machines, meaning the workload was not just parallel in threads, but truly on separate hardware. The **execution time** for the 10k-line job on EKS was similar to the Kind 4-partition run (within a similar few-second range). This is because the overhead of scheduling and starting pods in the cloud slightly adds latency, but the parallel processing time was equivalent.

| **Aspect** | **Naive Script** | **Local Spark Standalone** | **Kubernetes (Kind)** | **Kubernetes (AWS EKS)** |
| --- | --- | --- | --- | --- |
| **Deployment** | Single-process Python script | Docker Compose (Spark Master, Worker, Submit) | Spark on Kind (local K8s cluster) | Spark on AWS EKS |
| **Execution Time** | ~0.085 s | ~0.7 s (includes Spark overhead) | ~0.75 s (2 partitions), **~0.39 s** (4 partitions) | **~0.25 s** (fully utilized executors) |
| **Partition Count** | N/A (sequential processing) | 4 (explicitly set in code) | 2 (initial run), **4 (optimized)** | 4 |
| **Parallelism Achieved** | None | 2 concurrent tasks on single node | Initially 2/4 pods active → then 4/4 fully active | 4/4 pods active |
| **Executor Utilization** | N/A | 1–2 executors (based on core count) | 2/4 → 4/4 (with partition fix) | 4/4, each task well distributed |
| **Dynamic Allocation** | No | No | Yes (min 2, max 10 executors) | Yes (min 2, max 10 executors) |
| **Fault Tolerance** | None | Worker failure = job failure | Failed pods can restart | failed pods replaced automatically |

Analysis of Kubernetes mode with different Setting:

| **Metric / Phase** | **Default (no partition override)** | **Explicit 4 partitions** |
| --- | --- | --- |
| **Partition count** | 2 (Spark default) | 4 |
| **Executors allocated** | 2 (dynamic allocation initialExecutors=2) | 4 (scaled up by dynamic allocation) |
| **Stage 0 (reduceByKey)** | 2 reducers ⇒ ~0.50 s per task ⇒ **0.50 s** wall-clock | 4 reducers ⇒ ~0.50 s / 4 ≈ **0.13 s** |
| **Broadcast & scheduling overhead** | ~150 ms | ~150 ms |
| **Stage 1 (collect)** | 2 tasks ⇒ 72 ms + 74 ms (parallel) ⇒ **0.095 s** total | 4 tasks ⇒ 91–104 ms each (parallel) ⇒ **0.111 s** total |
| **End-to-end wall-clock** | ≈ 0.50 + 0.15 + 0.095 ≈ **0.745 s** | ≈ 0.13 + 0.15 + 0.111 ≈ **0.391 s** |
| **Executor utilization** | Both tasks ran on a single executor pod → other executor idle | One task per executor pod → all four busy |
| **Shuffle I/O per task** | ~31.8 KiB / 2 ≈ 16 KiB | ~31.8 KiB / 4 ≈ 8 KiB |

When running the Spark job on Kubernetes using Kind, we initially used the default 2 partitions, which resulted in only 2 executor pods doing work while the other 2 remained idle, leading to a longer execution time (~0.745 seconds). By explicitly setting 4 partitions, we matched the number of tasks to the available executors, enabling full parallelism. This improved resource utilization and reduced the overall runtime to ~0.391 seconds, clearly demonstrating that proper partitioning is crucial for maximizing efficiency in distributed processing.

1. **Future Work**

While the batch processing approach using Spark is effective, the next step is to **re-architect this batch job into a lightweight microservice-based system**. In a production environment where logs are continuously generated, it may be more useful to process usage data in real-time or on a rolling basis, rather than once per day in a big batch. Cron can be used as a scheduler for the same. By exposing the billing aggregation functionality as a service, we can provide RESTful endpoints for other parts of the system or even customers to query usage and cost.

1. **Conclusion**

Building the distributed billing aggregation system in stages demonstrated the trade-offs between simplicity and scalability. The **naive approach** was quick to implement but could not meet scalability and resilience requirements. Introducing Spark in a **local standalone cluster** added parallelism but was still limited to one machine’s resources. By moving to **Kubernetes (with Kind)**, we containerized the workload and gained the ability to allocate multiple executor pods, uncovering the importance of proper data partitioning to utilize those resources. Finally, running Spark on **AWS EKS** provided the full benefits of a cloud-native solution: speed through parallel execution, scalability by leveraging multiple nodes and dynamic resource allocation, cost-efficiency by using resources on-demand, and operational resilience thanks to Kubernetes and managed infrastructure. While testing we focused more on Local Kubernetes Cluster using Kind which we found was the best approach among the basic 3 variations. It could scale up to larger data where other two methods would falter.

1. **Collaboration**

**[1]** Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on

large clusters. Commun. ACM 51, 1 (January 2008), 107–113.

https://doi.org/10.1145/1327452.1327492

**[2]** Zaharia, Matei & Chowdhury, Mosharaf & Franklin, Michael & Shenker, Scott &

Stoica, Ion. (2010). Spark: Cluster Computing with Working Sets. Proceedings of the 2nd

USENIX conference on Hot topics in cloud computing. 10. 10-10.

**[3]** Changpeng, Zhu & Han, Bo & Zhao, Yinliang. (2022). A comparative performance

study of spark on kubernetes. The Journal of Supercomputing. 78. 10.1007/s11227-022-

04381-y.

Other than this, we went through official documentations:

* <https://spark.apache.org/docs/latest/api/python/index.html>
* <https://kind.sigs.k8s.io/>
* <https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/amazon-elastic-kubernetes-service.html>

We also used Stack Overflow and ChatGPT to debug and resolve different errors that we faced in our implementations. We have also used ChatGPT to write code for various small coding files, like adding logging, try-catch exceptions, comments, etc. We have used Grammarly and ChatGPT for rephrasing our final project report.