GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Master's Thesis

submitted in partial fulfillment of the
requirements for the course "M.Sc. Applied Data Science"

Comparative Analysis of On-Demand Distributed Computing Frameworks on Slurm-Based HPC Systems

---

Yasir Yasir

MatrNr: 29731672

First Supervisor:	Sadegh Keshtkar
Second Supervisor:	Jonathan Decker
First Reviewer:	Prof. Dr. Julian Kunkel

Georg-August-Universität Göttingen / GWDG
Institute of Computer Science (HPS Group)

August 30, 2026

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

Abstract

Data-science pipelines increasingly outgrow a single machine, pushing practitioners toward distributed computing frameworks such as Dask and Apache Spark on shared high-performance computing (HPC) infrastructure. Yet choosing between them is difficult because their relative performance is not fixed: it is governed by whichever resource bottleneck saturates first, be it input/output (I/O), central processing unit (CPU), shuffle, memory, or Basic Linear Algebra Subprograms (BLAS), and that bottleneck changes across the operations of a realistic pipeline. Existing comparisons typically rely on a single workload or a synthetic microbenchmark, leaving practitioners without a general basis for predicting which framework will win on their own pipeline. This thesis closes that gap by comparing Dask and Apache Spark (the two dominant general-purpose, Python-accessible engines) through a seven-workload benchmark suite deployed on demand inside Slurm allocations on this HPC infrastructure. The suite spans the data-science lifecycle (extract–transform–load (ETL), feature engineering, clustering at two difficulty levels, dense linear algebra, model training, and text vectorisation) and the resource-bottleneck taxonomy above. Six workloads use the ∼142-million-row New York City taxi dataset; one uses an Amazon review corpus. Both frameworks were evaluated across a factorial sweep of nodes, memory, and cores on the GWDG Scientific Compute Cluster, with Dask additionally run under Python 3.11 and 3.13.

No framework dominates. Spark, in both frameworks' default worker configurations, wins the four CPU- and shuffle-bound tabular workloads by 1.7×–2.7×, because its process-based executors parallelise Python-level work genuinely and its Catalyst optimiser fuses deep expression chains into a single physical pass, while Dask wins dense linear algebra by ∼17× and graphics processing unit (GPU)-accelerated term frequency–inverse document frequency (TF–IDF) by ∼10×, because its array-native design delegates directly to BLAS and avoids Spark's per-row Java virtual machine (JVM)-to-Python serialisation. Three further contributions sharpen this verdict: per-workload architectural explanations that tie each outcome to a specific internal mechanism, including thread-versus-process parallelism and the Global Interpreter Lock (GIL), query compilation, and shuffle-service maturity; an implementation-complexity evaluation showing, counter-intuitively, that Spark implementations are shorter and less complex, because MLlib and Catalyst absorb complexity into library calls; and practitioner decision support that conditions framework choice on workload class and resource envelope. The software environment is itself a first-class variable: Python 3.13 delivers 1.5×–2.1× gains over 3.11 for Dask. The result is a general, actionable heuristic: a practitioner can predict which framework will win a new workload by identifying where its computational bottleneck sits: in native array code, in Python-level execution, or in cross-language data movement, without re-running the benchmark suite.

Contents

List of Tables **iv**

List of Figures **v**

List of Abbreviations **vi**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Research Questions . . . 2
1.3 Contributions . . . 2
1.4 Structure of the Thesis . . . 3

2 Background **4**
2.1 Apache Spark Architecture . . . 4
2.2 Dask Architecture . . . 6
2.3 Slurm-Based HPC Clusters . . . 9
2.4 Python 3.11 vs 3.13: CPython Performance and the GIL . . . 10
2.5 The Data-Science Pipeline . . . 11
2.6 Adjacent and Newer Frameworks . . . 11

3 Related Work **12**
3.1 Distributed Frameworks on HPC . . . 12
3.2 Big-Data and Machine-Learning Benchmark Suites . . . 13
3.3 Prior Dask–Spark Comparisons . . . 13
3.4 GPU Acceleration in Distributed Analytics . . . 14
3.5 The JVM–Python Serialisation Boundary . . . 14
3.6 Deploying Data-Analytics Frameworks on Batch-Scheduled Clusters . . . 15
3.7 Interpreter-Level Performance and the Free-Threading Programme . . . 15
3.8 Benchmarking Methodology . . . 15
3.9 Positioning: The Related-Work Comparison Matrix . . . 16

4 Methodology **16**
4.1 Research Questions and Required Measurements . . . 16
4.2 Benchmark Suite Design . . . 17
4.2.1 Standing Up Ephemeral Clusters Inside a Slurm Allocation . . . 19
4.3 Benchmark Rationale: What Each Workload Tests . . . 20
4.4 Datasets . . . 23
4.5 Experimental Setup . . . 23
4.6 Benchmark Implementations . . . 26
4.7 Implementation-Complexity Evaluation Methodology . . . 28

5 Evaluation **28**
5.1 Overview of Measured Outcomes . . . 29
5.2 Data Transformation and Aggregation . . . 31
5.3 Feature Engineering . . . 33
5.4 K-means Clustering (Level 1) . . . 33

5.5 Gaussian Mixture Model / Expectation–Maximisation (Level 2) 34
5.6 Matrix Operations 35
5.7 Model Training: Linear Regression 36
5.8 Text Processing: TF–IDF 37
5.9 Python 3.11 versus 3.13: Effect on Dask Performance 38
5.10 Scalability 40
5.11 Architectural Mechanisms: Synthesis (**RQ3**) 43
5.12 Implementation Complexity (**RQ4**) 44

6 Discussion **48**
6.1 Summary of Findings 48
6.2 Implications 49
6.3 Limitations 53
6.4 Threats to Validity 53
6.5 Future Work 53

7 Conclusion **55**
7.1 Three Take-Home Messages 55
7.2 Data and Code Availability 55

References **56**

A Full Per-Configuration Results **A1**
A.1 K-means clustering (phase breakdown) A1
A.2 Gaussian mixture / expectation–maximisation (phase breakdown) A1
A.3 Data transformation (total and Dask scheduler breakdown) A2
A.4 Feature engineering, matrix operations, and model training (totals) A2
A.5 Resource metrics (selected) A3

B Implementation and Reproducibility Notes **A3**
B.1 Framework and environment versions A3
B.2 Spark executor sizing A4
B.3 Measurement protocol A4
B.4 Code basis A4

C Transparency Report: Tools and Third-Party Contributions **A4**
C.1 Overview A4
C.2 Per Section A5

List of Tables

1 Related-work comparison matrix. A plus sign (+) denotes that the criterion is addressed, a minus sign (−) that it is not, and a circle (○) that it is partially addressed. The rightmost column is this thesis. 16
2 Mapping from each research question to the measurement required to answer it. 17
3 Resource-bottleneck taxonomy and its coverage within the suite. 18
4 Per-benchmark rationale: what each workload tests, the Dask path, and the Spark path with the mechanism behind the result. 22
5 Hardware and partitions. Cells marked ‡ are GWDG SCC/NHR public-documentation defaults. 24
6 Parameter-sweep completion: Dask (Python 3.13, two-node slice shown). All cells completed except GMM at ≤64 GB (out-of-memory). 25
7 Parameter-sweep completion: Spark 3.5.1 (two-node slice shown). Same pattern: GMM is out-of-memory at ≤64 GB. 25
8 Measured wall-clock times (seconds) at best and worst configuration; Spark and Dask 3.13 shown. All tabular benchmarks use the full 142 M-row taxi dataset across eight hardware configurations. Best/worst are taken across all measured configurations. 29
9 TF–IDF wall-clock time and failure status, Spark CPU (MLlib) versus Dask GPU (cuML), across three corpus sizes. 38
10 Effect of Python version on Dask wall-clock time (mean across the two-node configurations). 39
11 Static code metrics (measured via `radon 6.0.1`). SLOC = source lines of code; LLOC = logical lines of code; CC = average cyclomatic complexity per function (A = 1–5, B = 6–10); MI = maintainability index (0–100, higher is better). 45
12 How each research question was answered. 48
13 Framework recommendation by workload class, derived from the mechanisms of Section 5. 52
14 Feature engineering, matrix operations, and model training: per-configuration wall-clock totals (seconds), two-node configurations. A2

List of Figures

1 Apache Spark deployment and execution architecture. 6
2 Dask deployment and task-graph execution. 8
3 Threads vs. processes and the GIL. 9
4 Workload coverage across the data-science lifecycle. 18
5 Benchmark stress matrix. 19
6 Allocation lifecycle for both deployment patterns. 21
7 Overall benchmark comparison (mean wall-clock time). 30
8 Winner heatmap, Dask 3.13 vs. Spark. 31
9 Data-processing phase breakdown, Dask 3.11 and 3.13. 32
10 Clustering phase breakdown. 34
11 Matrix-operations phase breakdown. 36
12 Linear regression, five repeated runs. 37
13 TF–IDF processing by phase, Dask GPU vs. Spark CPU. 38
14 Python 3.11 vs. 3.13 effect on Dask wall-clock time. 39
15 Scalability: wall-clock time vs. total core count. 41
16 Node-scaling efficiency, 2 to 4 nodes. 42
17 Memory sensitivity: 64 GB vs. 128 GB per node. 42
18 CPU topology: two nodes x 64 cores vs. four x 32. 43
19 Performance-complexity trade-off. 47
20 Framework-selection decision procedure. 51
21 Spark resource-utilisation panel across five tabular benchmarks. A3

List of Abbreviations

Abbreviation	Full form
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CC	Cyclomatic Complexity
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
EM	Expectation–Maximisation
ETL	Extract–Transform–Load
GC	Garbage Collection
GIL	Global Interpreter Lock
GMM	Gaussian Mixture Model
GPU	Graphics Processing Unit
GWDG	Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
HPC	High-Performance Computing
I/O	Input/Output
JVM	Java Virtual Machine
L-BFGS	Limited-memory Broyden–Fletcher–Goldfarb–Shanno (optimiser)
LLOC	Logical Lines of Code
MI	Maintainability Index
MLlib	(Spark's) Machine Learning Library
NLP	Natural Language Processing
OOM	Out-Of-Memory
PEP	Python Enhancement Proposal
RDD	Resilient Distributed Dataset
RQ	Research Question
SCC	Scientific Compute Cluster
SIMT	Single Instruction, Multiple Threads
SLOC	Source Lines of Code
SVD	Singular-Value Decomposition
TF–IDF	Term Frequency–Inverse Document Frequency
UDF	User-Defined Function

1 Introduction

1.1 Motivation

Python has become the *de facto* interface for distributed data processing and scientific computing [Zio+21; Aln+21]. Among the frameworks that expose such an interface, Apache Spark [Zah+16] and Dask [Roc15] are the two most established general-purpose engines that are both Python-accessible and deployable across a multi-node cluster (Section 2.6). Both present application programming interfaces (APIs) that closely mirror the single-machine pandas [McK10] and NumPy [Har+20] workflows that practitioners already know, and both can be deployed on demand inside a Slurm allocation on a modern HPC cluster without a standing service. This combination of familiar APIs and on-demand deployment is precisely what makes them attractive to data scientists who are not, and do not wish to become, distributed-systems engineers.

Despite their superficial similarity at the API level, the two frameworks rest on substantially different execution models. Spark is a Java-virtual-machine (JVM) based, query-plan-driven engine: its Catalyst optimiser rewrites a logical sequence of operations into a fused physical plan before any computation executes, and its Tungsten layer manages memory and code generation for that plan [Arm+15]. Dask, by contrast, is a Python-native task scheduler: it builds a directed acyclic graph (DAG) of lazy function calls and dispatches that graph to a set of worker processes, each of which by default runs many threads inside a single Python interpreter [Roc15]. These architectural asymmetries do not announce themselves at the API surface, yet they translate into performance differences that are large, non-obvious, and, crucially, strongly dependent on the workload.

The existing literature provides only limited and fragmentary empirical guidance for a practitioner who must choose between the two on Slurm-managed HPC infrastructure. Dugré et al. [Dug+19; Dug+23] compare the frameworks on input/output-bound neuroimaging pipelines and find broadly comparable runtime but a markedly higher memory footprint for Spark. Padulano et al. [Pad+23] demonstrate that both frameworks scale beyond two thousand cores for particle-physics analysis on a Slurm-managed CERN cluster, with cold Dask runs penalised by just-in-time compilation overhead. Databricks [Dat21] reports Spark to be two to twenty-five times faster than Dask on NYC-taxi tabular analytics, attributing the gap to Catalyst optimisation, whereas Rocklin [Roc23] shows Dask to be faster than Spark on TPC-H queries and able to complete twenty-one of twenty-two queries at the ten-terabyte scale where Spark fails all twenty-two. Zevnik et al. [Zev+22] find that Spark wins end-to-end machine-learning training while Dask is faster on DataFrame manipulation. Taken together, these studies establish a single robust meta-finding: *no framework dominates across workloads.* None, however, provides a systematic, multi-workload, multi-resource evaluation on Slurm-based HPC with a structured escalation of task complexity, and none evaluates *implementation complexity* alongside performance, despite adoption in practice being driven as much by developer ergonomics as by raw throughput.

A further dimension has emerged with recent CPython releases. Python 3.11 introduced a specialising adaptive interpreter (PEP 659; [Sha21]) that delivers substantial single-thread speed-ups, and Python 3.13 ships an experimental free-threaded build that makes the Global Interpreter Lock (GIL) optional (PEP 703; [Gro23]). Because Dask has historically relied on threading to extract intra-node parallelism, and because the GIL is

the central constraint on that strategy for Python-level code, the interpreter version is plausibly a first-order performance factor for Dask in a way that it is not for Spark, whose compute engine runs on the JVM. How much of a Dask benchmark result is attributable to the Python version, as distinct from the framework, is empirically unresolved at scale, and it interacts directly with the framework comparison.

This thesis addresses the combined gap. It develops a rigorous, reproducible, resource-factorial benchmark suite comparing Spark and Dask on Slurm-based HPC across the full data-science lifecycle; it explains the observed differences in terms of concrete architectural mechanisms grounded in the distributed-systems literature; it quantifies implementation effort per framework through static metrics, a structured qualitative assessment, and a catalogue of friction points; and it reports results under multiple Python versions, treating the software environment itself as a variable to be controlled rather than an incidental detail.

1.2 Research Questions

The overarching goal of this thesis is to give practitioners and researchers empirical, mechanistically grounded guidance for choosing between Dask and Apache Spark on Slurm-based HPC systems, across the full data-science lifecycle rather than for a single illustrative task. To make this goal tractable and answerable, it is decomposed into four research questions:

RQ1: Per-workload performance. For each workload class (input/output-bound, CPU-bound, shuffle-heavy, iterative-numerical, dense linear algebra, and text feature extraction), which framework executes faster under equivalent resource allocations, and by what margin? How does the ranking shift when task complexity is escalated from a base variant (Level 1) to a harder variant (Level 2) within the same category?

RQ2: Scalability. How does execution time scale with increasing CPU count, increasing node count, increasing per-node memory, and changing CPU topology (the same total core count distributed over fewer larger versus more smaller allocations)? Where does each framework exhibit diminishing returns or super-linear communication overhead?

RQ3: Architectural mechanisms. Can observed performance gaps be attributed to specific architectural properties of each framework (Catalyst query optimisation, task-graph depth, shuffle-service efficiency, and GIL effects under different Python builds), and how do resource-utilisation profiles (executor time, garbage-collection time, shuffle volume, memory pressure) differ between the frameworks for the same workload?

RQ4: Implementation complexity. How do equivalent Spark and Dask implementations compare in lines of code, cyclomatic complexity, maintainability, and structured difficulty? What migration effort is required to port a complete implementation from one framework to the other, and where do the largest impedance mismatches arise?

The first three questions are answered by the benchmark suite and its instrumentation; the fourth is answered by the implementation-complexity methodology of Section 4.7. Section 4 begins by mapping each research question to the specific measurement required to answer it (Table 2), so that the experimental design follows from the questions rather than the reverse.

1.3 Contributions

This thesis makes five contributions.

1. **A benchmark methodology for on-demand HPC frameworks.** A reproducible, Slurm-native methodology for comparing Spark and Dask across workloads and resource configurations, including cluster startup and teardown automation, parameter-sweep orchestration, and scheduler-level instrumentation. The suite is constructed deliberately to span two orthogonal design axes: the data-science lifecycle and the resource-bottleneck taxonomy (Section 4.2, Figures 4 and 5).

2. **An empirical multi-workload comparison.** A systematic Spark-versus-Dask benchmark on Slurm-based HPC covering seven workloads across the lifecycle, two complexity levels within the clustering category, a factorial parameter sweep over node count, memory, and cores, and multiple Python versions for Dask. To the author's knowledge this is the first study to combine all of these dimensions in a single, internally consistent evaluation.

3. **Architectural explanations.** A per-workload analysis that links each measured outcome to an internal mechanism: the thread-versus-process parallelism model and the GIL, the inversion of that relationship under BLAS, Catalyst whole-stage code generation, row-oriented versus array-native linear algebra, and parallel K-means initialisation overhead, grounded in primary sources on both frameworks and corroborated by phase-level timing and scheduler metrics.

4. **An implementation-complexity framework.** A replicable evaluation combining static code metrics (lines of code, cyclomatic complexity, maintainability index), a structured qualitative assessment, and a catalogue of impedance mismatches, applicable beyond this thesis to future framework comparisons. The framework yields a counter-intuitive finding: the framework with the simpler-*looking* API produces the longer and more complex *user code*, and it explains why.

5. **Practitioner decision support.** Concrete recommendations for framework selection conditioned on workload class, resource envelope, and team expertise, supported by measured trade-offs across all seven benchmark workloads (Section 6.2, Figure 20, Table 13).

1.4 Structure of the Thesis

The remainder of the thesis is organised as follows. **Section 2 (Background)** introduces the two frameworks' execution models, the Slurm-based HPC environment, the relevant CPython developments and the GIL, the data-science pipeline that motivates lifecycle-spanning benchmarking, and the adjacent frameworks that contextualise the choice of Dask and Spark. **Section 3 (Related Work)** surveys prior framework comparisons, benchmark suites, GPU-accelerated analytics, and benchmarking methodology, and positions this thesis against them in an explicit criteria-by-paper matrix. **Section 4 (Methodology)** derives the experimental design from the research questions, specifies the benchmark suite and the rationale for each workload, describes the datasets, the cluster and hardware, the parameter sweep, the timing protocol, and the implementation-complexity methodology. **Section 5 (Evaluation)** reports and analyses the measured results workload by workload, then treats scalability, the cross-cutting architectural mechanisms, and implementation complexity. **Section 6 (Discussion)** summarises the answers to the four research questions, draws out their implications, states the limitations, enumerates the threats to validity, and outlines the future work they motivate. **Section 7 (Conclusion)** revisits the contributions and their scope of validity and distils three take-home messages. References and two appendices of full per-configuration data

follow.

2 Background

This section establishes the technical foundation on which the rest of the thesis rests. It describes the execution models of Apache Spark and Dask in enough detail to make the later mechanistic explanations precise (Sections 2.1 and 2.2); the Slurm-managed HPC environment in which both are deployed on demand (Section 2.3); the CPython developments and the Global Interpreter Lock that make the Python version a first-class variable for Dask (Section 2.4); the data-science pipeline whose breadth motivates a lifecycle-spanning benchmark suite (Section 2.5); and the adjacent and newer frameworks against which the choice of Dask and Spark is positioned (Section 2.6).

2.1 Apache Spark Architecture

Apache Spark descends from the MapReduce model of large-scale data processing [DG04], which decomposed a computation into a *map* phase that applies a function to each record and a *reduce* phase that aggregates the results, with an intervening *shuffle* that redistributes data by key across the cluster. Spark generalised this model in two decisive ways. First, it introduced the Resilient Distributed Dataset (RDD), a fault-tolerant, partitioned, in-memory collection whose lineage of transformations can be recomputed on failure rather than reloaded from disk [Zah+12]; this made iterative algorithms, which repeatedly traverse the same data, far cheaper than on disk-oriented MapReduce. Second, it layered a declarative DataFrame and SQL interface on top of the RDD engine, processed by the Catalyst optimiser [Arm+15].

Catalyst is the component most responsible for Spark's behaviour on the tabular workloads in this thesis. A sequence of DataFrame operations is first represented as an unresolved logical plan, then resolved against the schema, optimised through a battery of rule-based and cost-based transformations (predicate pushdown, projection pruning, constant folding, and the reordering or fusing of operators), and finally compiled into a physical plan. The physical plan is executed by the Tungsten engine, which manages off-heap memory explicitly and, critically for the feature-engineering workload of this study, performs *whole-stage code generation*: a chain of narrow operators is compiled into a single tight loop of generated JVM bytecode rather than executed as a sequence of separate operator calls, a technique with its intellectual roots in the query-compilation approach of Neumann [Neu11]. The consequence is that a long chain of dependent column expressions can collapse, at planning time, into a single pass over the data.

Spark's runtime topology is a driver process that hosts the SparkContext and orchestrates execution, and a set of executor processes that perform the work. In the standalone cluster deployment used in this thesis, a master process manages a pool of worker daemons, and the application's executors are launched within the resources those workers expose. Each executor is a JVM that runs a fixed number of task slots; in this study executors are sized at four cores each. For native DataFrame and SQL operations the work happens entirely inside the JVM. For user-defined functions written in Python, including the `pandas_udf` used in the data-transformation workload, Spark forks a separate Python worker process per task slot and exchanges data with it over a serialisation boundary. This

last detail is central to the analysis in Section 5: because each Python user-defined function runs in its own operating-system process, no Python-level Global Interpreter Lock is shared across cores, and Python-level work therefore parallelises genuinely as the core count rises. The same boundary, however, imposes a serialisation cost whenever data must cross from the JVM to Python and back, which becomes the dominant cost in the row-oriented matrix workload of Section 5.6.

Spark's shuffle service is the final relevant component. When an operation requires data to be redistributed by key, such as a group-by aggregation, a join, or the per-iteration exchange of centroids in K-means, Spark writes shuffle blocks and serves them to the consuming stage. The maturity of this service is one reason Spark performs well on the shuffle-heavy iterative workloads in this study. Figure 1 summarises this deployment and execution topology.

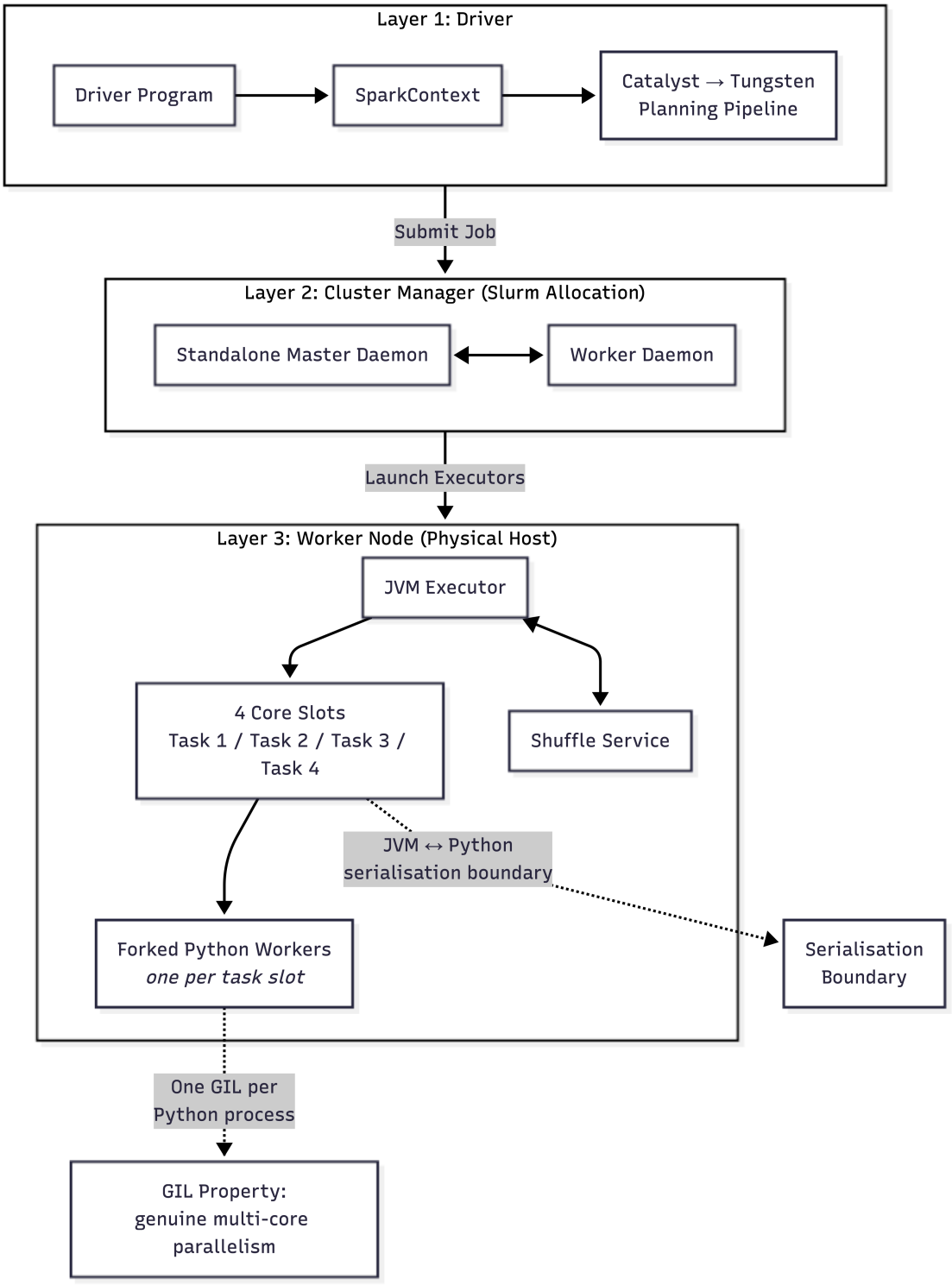


Figure 1: Apache Spark deployment and execution architecture: driver, Catalyst → Tungsten pipeline, Slurm-allocated executors, and the per-slot forked Python worker boundary.

2.2 Dask Architecture

Dask takes a fundamentally different route to the same goal of scaling Python computation across a cluster [Roc15]. Rather than presenting a declarative query interface that is

compiled and optimised, Dask exposes *parallel collections*: `dask.array`, `dask.dataframe`, and `dask.bag`, mirroring NumPy [Har+20] arrays, pandas [McK10] DataFrames, and Python iterables respectively, together with a low-level `delayed` interface for arbitrary task graphs. Operations on these collections are lazy: they build a directed acyclic graph in which nodes are Python functions and edges are data dependencies. When a result is requested, the graph is handed to a scheduler that executes the tasks, respecting dependencies and moving intermediate data between workers as needed.

The two architectural facts that matter most for this thesis are the nature of Dask's collections and the nature of its default worker. First, `dask.array` is a genuine extension of NumPy: a large array is represented as a grid of smaller NumPy *chunks*, and an operation such as a matrix multiplication or a singular-value decomposition is expressed as a blocked algorithm over those chunks, each block of which is executed by the corresponding NumPy and BLAS routine. This is why Dask is so effective on the dense-linear-algebra workload of Section 5.6: the heavy arithmetic is delegated directly to optimised BLAS kernels, with Dask responsible only for the blocking and the reduction. The BLAS backend behind NumPy in both worker environments is `scipy-openblas` (the OpenBLAS build vendored by SciPy's wheels), for both the Python 3.11 and 3.13 builds. The machine-learning library `dask_ml` provides distributed estimators with a scikit-learn-style `fit`/`transform` interface on top of these collections.

Second, Dask's default deployment runs a single worker process per node, and that worker uses many threads to execute tasks concurrently. The deployment scripts used in this thesis specify neither an explicit worker count nor a thread count, so each worker autosizes its thread pool to the cores in the Slurm allocation; the scheduler logs confirm one worker per node with thirty-two or sixty-four threads each. Because those threads share a single Python interpreter, they share a single Global Interpreter Lock. The consequences of this choice cut in opposite directions depending on the workload, and reconciling those two directions is the central analytical thread of Section 5. For Python-level loops the GIL serialises execution, so additional threads yield little speed-up; for NumPy and BLAS kernels, which release the GIL for the duration of their C or Fortran computation [Har+20], the same threads parallelise fully. The thread model is therefore simultaneously Dask's greatest liability and its greatest asset. Figure 2 summarises this task-graph and thread-pool deployment, and Figure 3 contrasts it directly with Spark's per-task forked-process model.

A scheduler with this design pays a per-task coordination overhead (the cost of constructing the graph, dispatching tasks, and tracking dependencies and data locations) that has been characterised independently by Böhm and Beránek [BB20]. For workloads dominated by a few large compute kernels this overhead is negligible; for workloads composed of many small tasks, such as an iterative algorithm with parallel initialisation, it can dominate, as the K-means analysis of Section 5.4 shows.

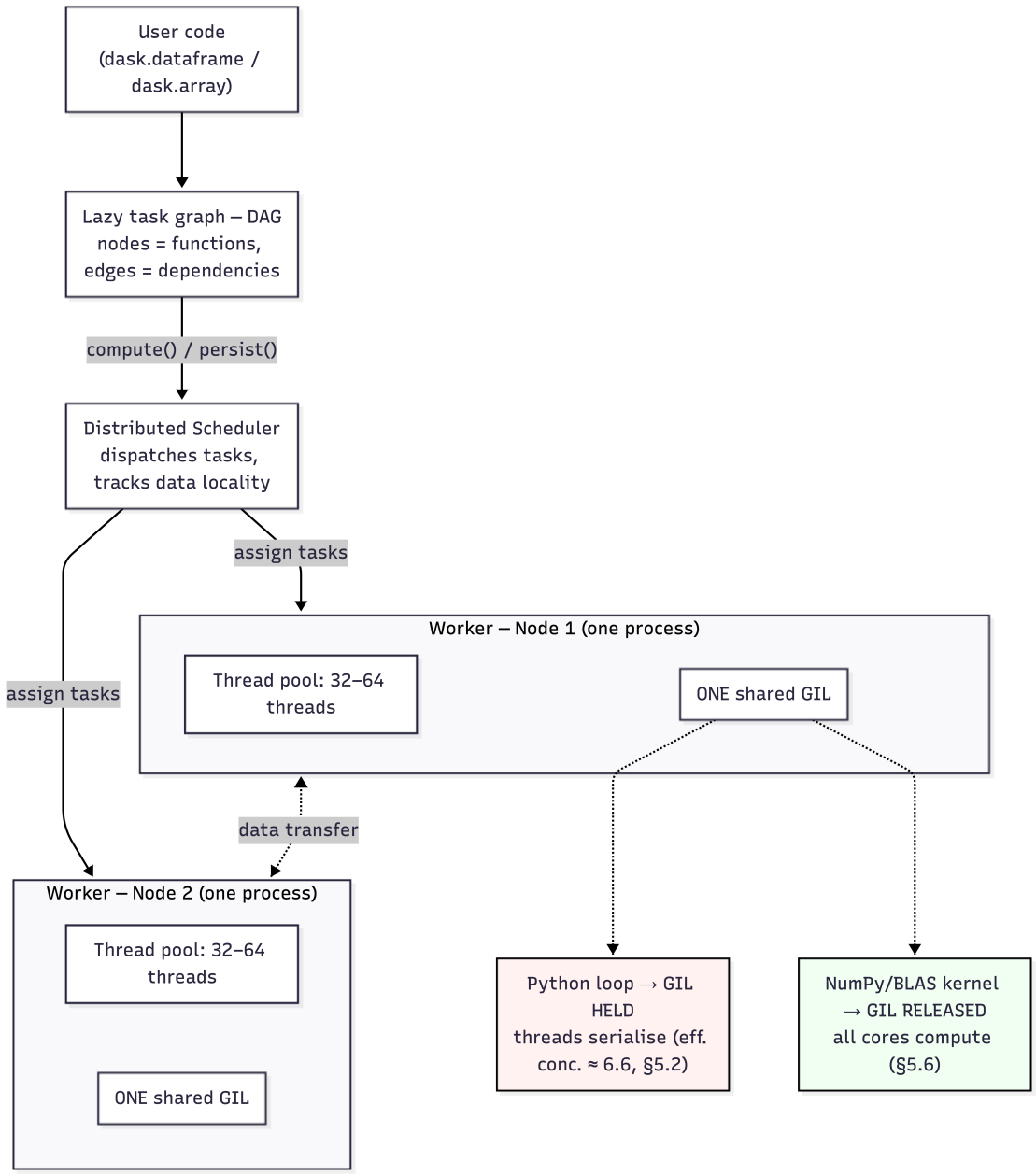


Figure 2: Dask deployment and task-graph execution. The client builds a lazy DAG; the distributed scheduler dispatches tasks to one worker process per node, each with a thread pool sized to the allocation and sharing a single GIL. The GIL is held during Python-level loops (serialising them) and released during NumPy/BLAS kernels (which then parallelise fully).

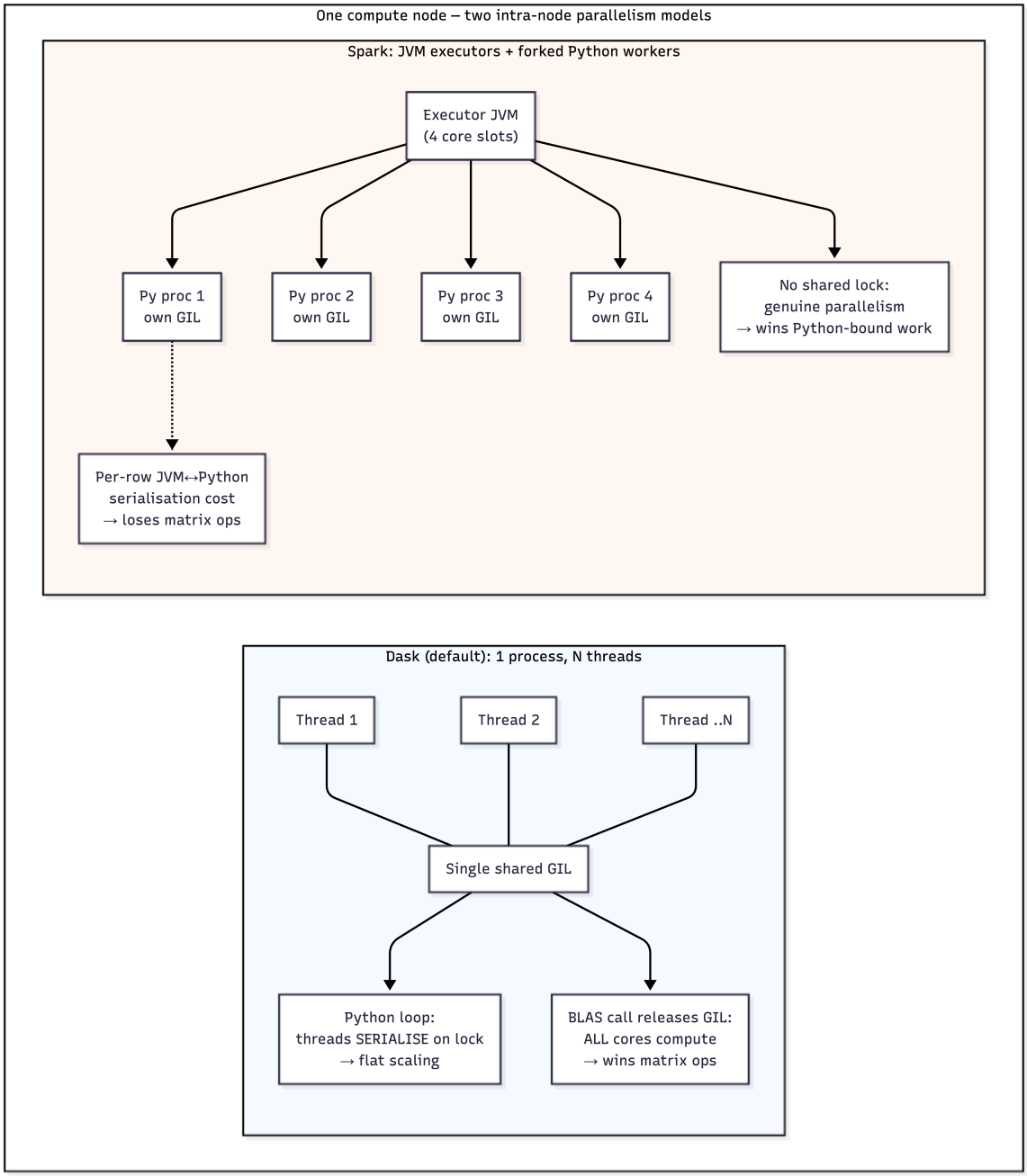


Figure 3: Thread-based versus process-based intra-node parallelism and the GIL. *Left (Dask):* one Python process with N threads sharing a single GIL; a Python loop serialises on the lock while a BLAS call releases it and runs on all cores. *Right (Spark):* JVM executors with four-core task slots, each Python UDF forked into its own process with its own GIL, giving genuine parallelism at the cost of a per-row serialisation boundary.

2.3 Slurm-Based HPC Clusters

High-performance computing clusters of the kind operated by the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG) differ from the cloud-managed clusters on which Spark and Dask are most commonly benchmarked, e.g. Databricks' own Koalas/PySpark-versus-Dask comparison [Dat21]. Resources are not provisioned as a standing, elastic service; they are requested from a batch scheduler, here Slurm [YJG03],

which allocates a set of nodes, cores, and memory to a job for a bounded wall-clock time and then reclaims them. Neither Spark nor Dask runs as a persistent cluster on such a system. Instead, each must be *deployed on demand* within a Slurm allocation: the framework's scheduler or master and its workers or executors are started across the allocated nodes at the beginning of the job and torn down at the end. This on-demand deployment model is the operating context for the entire thesis, and it is the reason that cluster startup and teardown automation is treated as a first-class component of the methodology (Section 4.5).

The GWDG Scientific Compute Cluster (SCC) [GWD26] is a heterogeneous system whose partitions expose different hardware. The relevant partitions for this study are a GPU partition (containing NVIDIA A100 accelerators alongside AMD EPYC host processors) and a CPU partition (containing Intel Xeon processors). Nodes within a partition are connected by a high-bandwidth, low-latency InfiniBand fabric, and a shared parallel filesystem holds the input datasets, with node-local scratch storage available for framework temporary files and logs. The specific node specifications used in this thesis are stated in the experimental setup (Section 4.5, Table 5).

The on-demand model has a methodological virtue that this study exploits deliberately: because each framework's cluster is started fresh for every configuration, runs all of its benchmarks, and is then torn down before the next configuration begins, there is no possibility of state (cached data, warmed JVMs, accumulated scheduler state) leaking from one configuration into another. Every configuration is measured from a clean start.

2.4 Python 3.11 vs 3.13: CPython Performance and the GIL

The reference implementation of Python, CPython, has historically executed Python bytecode through a single interpreter loop protected by the Global Interpreter Lock, a mutex that permits only one thread to execute Python bytecode at a time [Mon26]. The GIL simplifies the interpreter and the C extension interface, but it means that pure-Python code cannot use more than one core within a single process, regardless of how many threads are launched. C extensions, including NumPy and the BLAS libraries beneath it, may release the GIL while executing native code, which is what allows numerical kernels to use multiple cores even within Dask's single-process, multi-threaded worker [Har+20]. This single distinction explains why, throughout Section 5, Dask's thread model behaves so differently on Python-level loops than on BLAS-backed array operations.

Two recent CPython developments bear directly on this thesis. Python 3.11 introduced a *specialising adaptive interpreter* (PEP 659; [Sha21]), which observes the types and operations encountered at each bytecode and rewrites hot instructions into specialised, faster forms; the result is a broad single-thread speed-up over earlier versions, with no change to the threading model. Python 3.13 ships an experimental *free-threaded build* (PEP 703; [Gro23]) in which the GIL can be disabled, in principle allowing pure-Python code to run on multiple cores within one process. Because Dask's intra-node parallelism rests on threads, both developments are plausibly first-order performance factors for Dask: the adaptive interpreter speeds up the Python-level scheduling and compute that the GIL otherwise serialises, and free-threading could in principle change the effective concurrency of Python-level loops altogether.

The same developments are *not* expected to be first-order factors for Spark. Spark's

compute engine, comprising Catalyst, Tungsten, the shuffle service, and MLlib, runs on the JVM [Dün+17; Arm+15]; the Python process in a PySpark application acts only as a driver and as a host for user-defined functions, and is therefore not on the engine's compute critical path. This asymmetry (the interpreter version is a direct performance factor for Dask but not for Spark) is the justification, developed in Section 4.5, for benchmarking Dask under two Python versions while holding Spark at one. It also makes the Python version a confound that any honest Dask benchmark must control: a result reported for "Dask" without stating the Python version, and indeed without pinning the full surrounding library stack, is under-specified.

2.5 The Data-Science Pipeline

A recurring observation in the machine-learning-systems literature is that the model-training code which receives the most research attention constitutes only a small fraction of a real production system; the surrounding infrastructure, comprising data collection, extract–transform–load, feature engineering, verification, and serving, dominates both the codebase and the operational effort [Scu+15]. This observation has a direct methodological consequence for a framework comparison. If one benchmarks only model training, one measures the frameworks on the slice of the lifecycle that is least representative of where practitioners actually spend their time and compute. A comparison that aims to inform real framework-selection decisions must therefore span the lifecycle: bulk ETL and aggregation, feature engineering, unsupervised and supervised learning, dense numerical work, and — where the data are unstructured — natural-language processing.

The suite places at least one workload on each stage of a representative data-science pipeline. Difficulty also escalates *within* a stage, via the Level-1 (K-means) to Level-2 (Gaussian mixture) pair in the clustering category, and this shows how each framework behaves as algorithmic intensity rises while the data scale is held constant.

2.6 Adjacent and Newer Frameworks

Dask and Spark are not the only options for scaling Python data work, and a complete picture requires positioning them against adjacent systems, if only to make clear why they, and not the alternatives, are the subject of this thesis. DuckDB [RM19] is an embeddable, single-node, vectorised analytical database that has become extremely competitive on medium-sized analytical queries; it is, however, fundamentally single-node and therefore addresses a different regime from the distributed, multi-node HPC setting studied here. Ray [Mor+18] is a Python-native, actor-based distributed framework aimed at reinforcement learning and general distributed applications; it overlaps with Dask in ambition but targets task- and actor-parallelism rather than the array- and DataFrame-parallel analytics that this suite stresses. Polars is a Rust-backed DataFrame library with a lazy, query-optimised execution model that rivals Catalyst on single-node tabular work, with distributed execution comparatively nascent.

These systems are best understood as the context that frames the present question rather than as competitors within it. The thesis asks specifically how *distributed Dask and Spark* compare when deployed on demand on Slurm-based HPC, the regime in which a workload is too large for a single node and the available infrastructure is a batch-scheduled cluster rather than a managed cloud service. DuckDB and Polars answer the single-node question; Ray answers a different distributed question; Dask and Spark are the

two mature, general-purpose, distributed, Python-accessible analytics engines for which the HPC-deployment comparison is both open and practically consequential.

To justify Ray's exclusion in more depth: Ray [Mor+18] is built around a low-level task and actor abstraction rather than a collections API: its scheduler is decentralised and its object store (Plasma) holds immutable shared-memory objects, a design targeted at the fine-grained, heterogeneous task graphs of reinforcement learning and model serving rather than at bulk DataFrame or array analytics. Ray Data has since added a dataset abstraction, but its data-analytics layer remains younger than either Spark's or Dask's, and, decisively for this thesis, Ray occupies a different point in the design space: it answers "how do I scale arbitrary Python tasks and stateful actors," not "how do I scale a DataFrame/array data-science pipeline," which is the question this benchmark suite operationalises (Section 4.2). Including it would have required either restricting the suite to its intersection with Ray Data or hand-writing actor-level implementations that no practitioner would deploy, both of which would break the like-for-like implementation-complexity comparison of **RQ4**.

3 Related Work

This section surveys the prior work against which the thesis is positioned, organised into five strands: framework comparisons on HPC (Section 3.1), big-data and machine-learning benchmark suites (Section 3.2), prior Dask–Spark comparisons specifically (Section 3.3), GPU-accelerated analytics (Section 3.4), and the benchmarking-methodology literature that underpins the measurement protocol (Section 3.5). It concludes with an explicit criteria-by-paper matrix that locates the contribution of this thesis (Section 3.6).

3.1 Distributed Frameworks on HPC

The most directly comparable prior studies evaluate Dask and Spark on genuine HPC infrastructure. Dugré et al. [Dug+19] compare the two frameworks on data-intensive neuroimaging pipelines, and the follow-up journal study [Dug+23] extends this to HPC systems specifically; their central findings are that runtime is broadly comparable on input/output-bound pipelines but that Spark carries a substantially higher memory footprint, and that data movement across the Java-to-Python boundary is a recurring cost for PySpark. The latter observation is one of the empirical anchors for the matrix-operations analysis of Section 5.6 in this thesis. Padulano et al. [Pad+23] study large-scale data analysis for high-energy physics on a CERN HPC cluster managed by Slurm, demonstrating that both frameworks scale beyond two thousand cores and characterising the cold-start penalty that Dask incurs from just-in-time compilation overhead. These studies establish that the HPC-on-Slurm regime is a legitimate and distinct setting for framework comparison, but each is confined to a narrow workload domain (neuroimaging or particle physics) rather than spanning the data-science lifecycle, and none addresses implementation complexity or the Python-version question.

3.2 Big-Data and Machine-Learning Benchmark Suites

The design of the benchmark suite in this thesis draws on the established big-data benchmarking tradition. HiBench [Hua+10] is a widely used suite characterising the MapReduce-based data-analysis workloads of the Hadoop and Spark ecosystems, comprising micro-benchmarks (sort, word-count, scan) and more complex workloads (K-means, Bayesian classification, PageRank). BigDataBench [Wan+14] is a broader suite drawn from internet-service workloads, spanning structured, semi-structured, and unstructured data and including search, social-network, and e-commerce scenarios. Both suites motivate the specific workloads chosen here: scan-and-aggregate tabular processing, K-means clustering, and TF–IDF-based text feature extraction all have direct precedent in them. Neither suite, however, includes Dask or any Python-native comparison path; both are oriented to the JVM ecosystem. This gap (standardised workloads exist, but not a cross-framework suite that places Dask and Spark on equal footing) is part of what this thesis addresses.

Beyond the Hadoop-era suites, the decision-support benchmarks of the Transaction Processing Performance Council, TPC-H and TPC-DS, remain the de facto standard for tabular engines, and are the basis of Rocklin's Dask-versus-Spark comparison [Roc23]. Their queries, however, are pure SQL-style analytics: they exercise scan, join, and aggregation but contain no iterative machine learning, no dense linear algebra, and no text vectorisation, and therefore cover only one of the two axes this thesis spans. The suite built here can be read as a complement to TPC-style benchmarking: narrower in SQL sophistication, but broader across the lifecycle and bottleneck taxonomy that determine framework choice for data-science work.

3.3 Prior Dask–Spark Comparisons

Several studies and industry reports compare Dask and Spark directly, and their collective message is that neither framework dominates. Zevnik et al. [Zev+22] compare the frameworks on image-classification workloads and find that Spark wins on end-to-end machine-learning training while Dask is faster on DataFrame manipulation. The Databricks engineering blog [Dat21] reports Spark to be between two and twenty-five times faster than Dask on NYC-taxi tabular analytics, attributing the advantage to Catalyst optimisation. Rocklin [Roc23], presenting Dask's own benchmarking on the TPC-H decision-support queries, reaches the opposite-leaning conclusion: Dask is faster than Spark on many of those queries and substantially more robust at extreme scale, completing twenty-one of twenty-two queries at ten terabytes where Spark fails all twenty-two. Both sources are party-interested and non-peer-reviewed: Databricks is Spark's principal commercial steward, and Rocklin is Dask's original author. Their opposing conclusions are therefore best read as bounding claims rather than neutral measurements. The apparent contradiction between the Databricks and Rocklin results is itself instructive, since it reflects the dependence of the verdict on the workload, the data scale, and the configuration. It is precisely the kind of single-number, single-workload claim that this thesis seeks to replace with a structured, multi-workload, mechanistically explained comparison. None of these prior comparisons is conducted on Slurm-based HPC, varies the Python version, or measures implementation complexity.

3.4 GPU Acceleration in Distributed Analytics

The text-processing workload in this thesis is a deliberate comparison between a GPU-accelerated Dask path and a CPU Spark path, and it rests on a body of work on GPU-accelerated analytics. The RAPIDS ecosystem, of which cuDF, CuPy [Oku+17], and cuML are part, provides GPU-accelerated, largely drop-in replacements for pandas, NumPy, and scikit-learn [RPN20]; cuML in particular supplies the GPU hashing vectoriser and TF–IDF transformer used in the Dask text pipeline. The expectation that this path will deliver an order-of-magnitude speed-up over a CPU implementation is grounded in Erra et al. [Err+15], who report large speed-ups for GPU-accelerated TF–IDF over massive text streams, exploiting the embarrassingly parallel, per-document nature of hashing-based vectorisation. This thesis confirms the direction and magnitude of that expectation with its own measurements, and additionally reports a robustness dimension, namely the CPU path's failure on larger corpora, that the speed-focused prior work does not emphasise.

The RAPIDS ecosystem [RAP18] is not excluded from this thesis so much as *scoped*: the TF–IDF workload (Section 5.8) deliberately pairs a Dask-orchestrated cuDF/CuPy path against Spark's CPU path, measuring exactly the boundary where the GPU stack pays off ($\sim$10$\times$). A full RAPIDS-versus-Spark suite would confound the framework variable with the accelerator variable across all seven workloads; holding the GPU to one workload isolates it as a controlled contrast instead. The RAPIDS Accelerator for Apache Spark (`spark-rapids`) [NVI20] closes the loop from the other side, bringing GPU acceleration *inside* Spark's SQL engine, and is named in Section 6.5 as the natural extension of the TF–IDF result.

3.5 The JVM–Python Serialisation Boundary

A recurring cost in Python-accessible JVM systems is the movement of data between the JVM and the Python interpreter. Early PySpark exchanged rows via pickle serialisation, a per-row cost that Dünner et al. [Dün+17] identify as a first-order factor in distributed machine-learning performance on Spark. The introduction of Apache Arrow [Apa16][1] as a columnar interchange format, and of vectorised `pandas_udf` user-defined functions built on it, reduced this cost substantially for columnar paths by amortising serialisation over batches rather than rows, extending the Spark SQL execution model of Armbrust et al. [Arm+15]. Crucially, however, the row-oriented `RowMatrix` path in MLlib predates and bypasses this columnar route, which is precisely the boundary probed by the matrix workload of Section 5.6. Dugré et al. [Dug+19; Dug+23] likewise single out Java-to-Python data movement as a recurring cost for PySpark in neuroimaging pipelines. This literature predicts the direction of the matrix-operations result before it is measured: where a workload can stay columnar, the boundary is cheap; where it forces row-at-a-time crossing, the boundary dominates.

[1] A cross-language, in-memory columnar data format that lets systems exchange tabular data without a per-row serialisation step.

3.6 Deploying Data-Analytics Frameworks on Batch-Scheduled Clusters

A small body of tooling and experience reports addresses the mechanics of running elastic analytics frameworks under batch schedulers. On the Dask side, `dask-jobqueue` provides first-class Slurm (and PBS/LSF) cluster classes that submit workers as batch jobs, and Dask's own `dask.distributed` deployment supports the `srun`-launched pattern used in this thesis [Roc15]. On the Spark side, no official Slurm integration exists; practitioners rely on site-specific launcher scripts that stand up a standalone master and workers inside an allocation, the approach taken here and by Padulano et al. [Pad+23] at CERN. This asymmetry is itself a finding of the deployment literature: Dask's HPC deployment story is institutionalised in maintained tooling, whereas Spark's remains ad hoc, a difference that surfaces in this thesis as the resource-configuration impedance mismatch of Section 5.12 (manual executor sizing on Spark versus automatic allocation inheritance on Dask).

3.7 Interpreter-Level Performance and the Free-Threading Programme

The faster-CPython programme has produced a sequence of interpreter-level improvements whose interaction with distributed frameworks is only beginning to be studied. The specialising adaptive interpreter of PEP 659 [Sha21] delivers broad single-thread speed-ups from Python 3.11 onward, and PEP 703 [Gro23] makes the GIL optional in the experimental free-threaded 3.13 build. Early community benchmarks of free-threaded CPython report meaningful multi-threaded scaling for pure-Python workloads at the cost of some single-thread overhead, but no prior work measures these interpreter effects *through* a distributed framework at cluster scale. This thesis contributes exactly that measurement for Dask (Section 5.9), bringing the measurement-bias concerns of Mytkowicz et al. [Myt+09] into the Python-versioning setting.

3.8 Benchmarking Methodology

Because a central concern of this thesis is the credibility of its own measurements, the benchmarking-methodology literature is treated as related work in its own right. Georges et al. [GBE07] set out the requirements for statistically rigorous performance evaluation on the JVM: discarding warm-up runs, measuring in steady state, and reporting dispersion rather than single numbers, all of which are reflected in the repeated-run protocol of Section 4.5 and in the explicit treatment of cold-start runs in the model-training analysis of Section 5.7. Mytkowicz et al. [Myt+09] demonstrate how easily measurement bias creeps into performance evaluation through factors as incidental as linking order and environment-variable size, a warning that motivates the clean-start, fresh-cluster-per-configuration discipline of this study. Raasveldt and Mühleisen [RM18] catalogue common pitfalls in database benchmarking, among them failing to control for caching and comparing systems on subtly different tasks, which directly informs the framework-parity principle and the explicit flagging of the two deliberate divergences (the GPU TF–IDF path and the library-versus-custom GMM).

3.9 Positioning: The Related-Work Comparison Matrix

Table 1 locates the contribution of this thesis against the most relevant prior work along seven criteria. The pattern in the table is the argument: each prior study addresses a subset of the dimensions, and several dimensions (most notably the Python-version effect, the implementation-complexity evaluation, and the structured Level-1-to-Level-2 difficulty escalation) are addressed by none of them. No prior study covers multi-workload breadth, scalability, HPC/Slurm deployment, the Python-version effect, implementation complexity, a GPU path, and intra-category difficulty escalation simultaneously. That intersection is the gap this thesis fills.

Table 1: Related-work comparison matrix. A plus sign (+) denotes that the criterion is addressed, a minus sign (−) that it is not, and a circle (○) that it is partially addressed. The rightmost column is this thesis.

Criterion	Dugré '19/'24	Padulano '23	Zevnik '22	Databricks '21	Rocklin '23	Erra '15	This thesis
Multi-workload breadth	−	−	○	−	+	−	+
Scalability analysis	+	+	○	−	+	−	+
HPC / Slurm deployment	○	+	−	−	−	−	+
Python-version effect	−	−	−	−	−	−	+
Implementation complexity	−	−	−	−	−	−	+
GPU acceleration path	−	−	−	−	−	+	+
L1 → L2 difficulty escalation	−	−	−	−	−	−	+

4 Methodology

This section specifies how the research questions are answered. It begins from the questions themselves and the measurements each requires (Section 4.1), and then introduces the benchmark suite as the instrument that produces those measurements (Sections 4.2–4.3), the datasets (Section 4.4), the experimental platform and hardware (Section 4.5), the seven benchmark implementations (Section 4.6), and the implementation-complexity methodology that answers **RQ4** (Section 4.7).

4.1 Research Questions and Required Measurements

Each question fixes what must be measured, which in turn determines the benchmark suite and the experimental setup described in the rest of this section. The benchmark suite of Section 4.2 is the instrument that produces the measurements required by **RQ1**–**RQ3**;

the implementation-complexity methodology of Section 4.7 produces the measurements required by **RQ4**. Table 2 makes this mapping explicit.

Table 2: Mapping from each research question to the measurement required to answer it.

RQ	Question	Measurement required
RQ1	Which framework is faster per workload, and by how much?	Wall-clock time per workload at matched resource allocations, with repeated runs and reported dispersion; per-phase timing to localise the cost.
RQ2	How does runtime scale with resources?	Wall-clock time as a function of CPU count, node count, per-node memory, and CPU topology.
RQ3	Can performance gaps be attributed to architectural mechanisms?	Phase-level timing plus scheduler and executor metrics (cumulative compute time, garbage-collection time, shuffle volume, iteration counts) linking each gap to a specific mechanism.
RQ4	How do implementations compare in complexity?	Static code metrics (lines of code, cyclomatic complexity, maintainability index), a structured qualitative assessment, and an impedance-mismatch catalogue.

4.2 Benchmark Suite Design

The suite is constructed to span two orthogonal design axes.

Axis A: coverage of the data-science and machine-learning lifecycle. As argued in Section 2.5, model training is only a small part of a real pipeline [Scu+15]; ETL and feature engineering dominate real effort. The suite therefore places at least one workload on each stage of a representative pipeline (Figure 4): raw ETL and aggregation, feature engineering, unsupervised learning (at two difficulty levels), dense numerical linear algebra, supervised model training, and — for unstructured data — natural-language feature extraction. This ensures that conclusions generalise across the activities a realistic workload performs.

Axis B: coverage of the resource-bottleneck taxonomy. Because distributed-framework performance is determined by the first resource to saturate, the suite is also designed to exercise each principal bottleneck class (Figure 5). The classes, and the workloads in which each is primarily exercised, are summarised in Table 3.

Table 3: Resource-bottleneck taxonomy and its coverage within the suite.

Bottleneck class	Where it is exercised
Input/output-bound	Columnar Parquet reads at load, in every benchmark; dominant during the load phase of data transformation.
Central-processing-bound	A 500-iteration trigonometric kernel (data transformation) and a 100-iteration refinement loop (feature engineering).
Shuffle / communication-bound	The group-by aggregation (data transformation); the centroid all-reduce (K-means); the component all-reduce (Gaussian mixture); the gradient all-reduce (regression).
Memory-bound	The Gaussian-mixture responsibility matrix (≈11 GB at 142 M rows × 10 components); the full in-memory tabular frame.
Dense numerical / BLAS	K-means, the Gaussian-mixture maximisation step, the Gram matrix and singular-value decomposition of the matrix benchmark, and the regression gradient.

The framework-parity principle. A third design principle underpins the suite: each benchmark is implemented with identical semantics on both engines, so that measured differences are attributable to the frameworks rather than to differing problem definitions. There are exactly two deliberate exceptions, both documented as threats to validity (Section 6.4): the text-processing workload compares a GPU Dask path against a CPU Spark path by design, and the Gaussian-mixture workload compares a hand-written Dask expectation–maximisation routine against Spark's library implementation. These are methodological choices made to expose ecosystem-level differences, not parity defects.

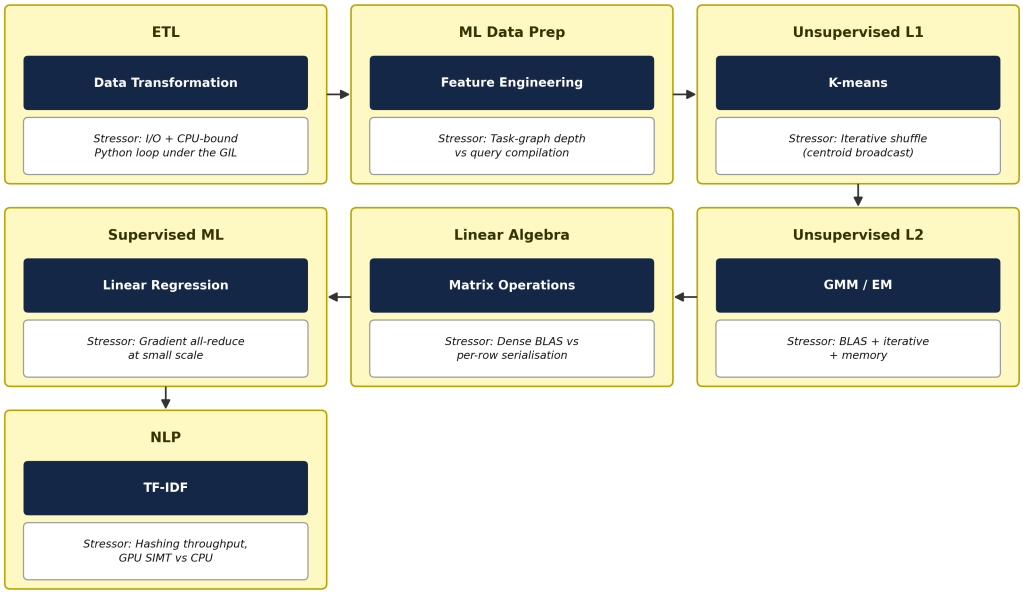


Figure 4: Workload coverage across the data-science lifecycle. Each stage carries one representative benchmark, labelled with its dominant resource stressor.

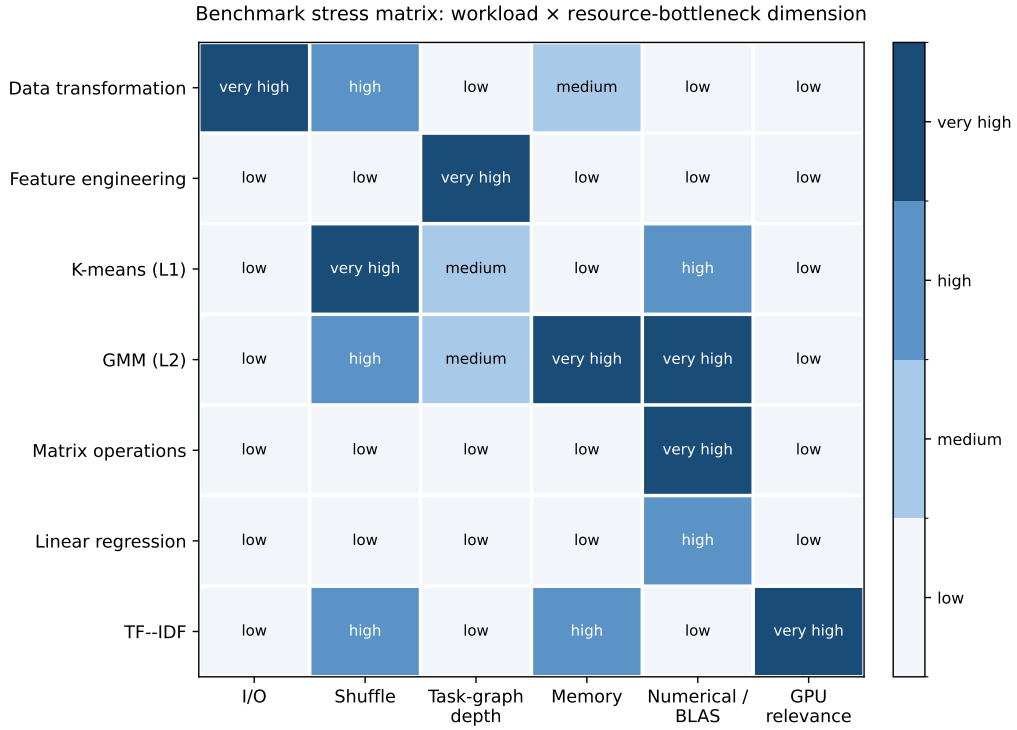


Figure 5: Benchmark stress matrix. A 7 × 6 heatmap: rows are the seven workloads, columns are I/O, shuffle, task-graph depth, memory, numerical/BLAS, GPU relevance, cells coloured by qualitative intensity (low/medium/high/very high). This visualises Axis B and shows that every resource class is a primary stressor in at least one workload.

4.2.1 Standing Up Ephemeral Clusters Inside a Slurm Allocation

Neither framework is a native Slurm citizen: both assume a long-lived cluster, while Slurm grants a bounded allocation. The benchmark automation therefore treats a cluster as a *disposable artefact of one allocation* (created after `salloc`/`sbatch` grants nodes, destroyed before the allocation ends), and the two frameworks require markedly different choreography to achieve this.

Spark (standalone-in-Slurm). Within the allocation, the job script (1) selects the first allocated node as master and starts `start-master.sh`, binding to `spark://<node>:7077`; (2) launches one worker daemon per remaining node via `srun -nodes=1 -exact` wrapping `start-worker.sh`, each registering with the master over the InfiniBand fabric; (3) waits for worker registration by polling the master's REST endpoint (port 8080) until the expected core count appears; (4) submits the application with `spark-submit`, sizing executors at four cores each (Section 2.1); (5) on completion or timeout, tears down workers then master, so that no daemon outlives the allocation. The failure modes observed during the campaign were port collisions when two allocations landed on a previously shared node (mitigated by deriving ports from `$SLURM_JOB_ID`) and workers registering before the shared filesystem mount was warm (mitigated by the registration poll in step 3).

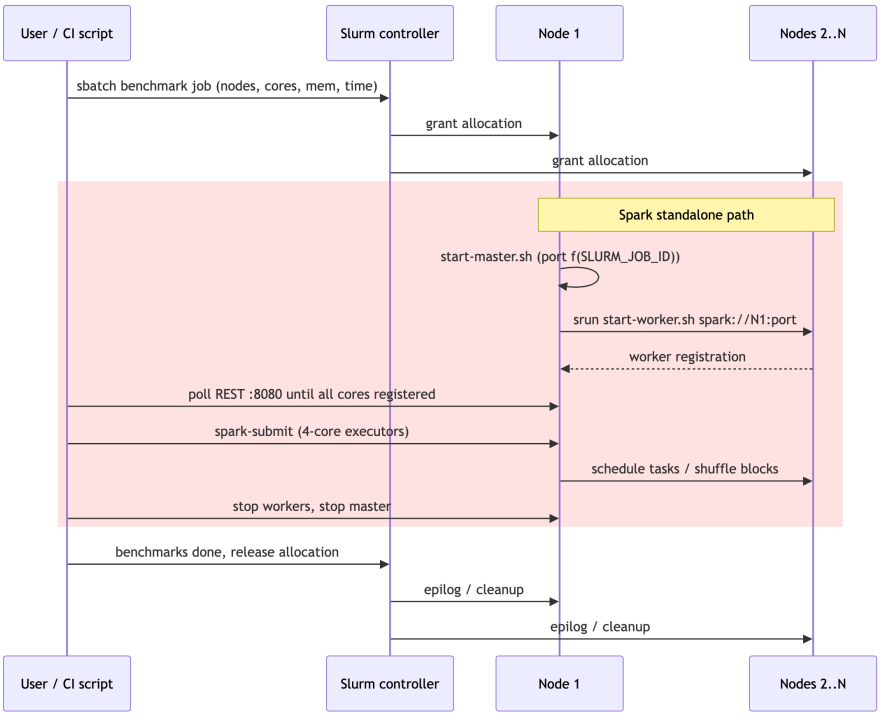
Dask (`dask.distributed` via srun). The pattern is lighter: (1) `dask scheduler` starts on the first node, writing its address to a scheduler file on the shared filesystem; (2) workers start on all nodes via a single `srun` broadcast, each reading the scheduler file (no per-node scripting); (3) the client connects through the same file, making the rendezvous

filesystem-mediated rather than port-mediated; (4) teardown is `client.shutdown()` plus the natural death of `srun` children at allocation end. Observed failure modes were stale scheduler files from a previous crashed run (mitigated by job-unique file names) and, on the GPU partition, CUDA context initialisation serialising worker start.

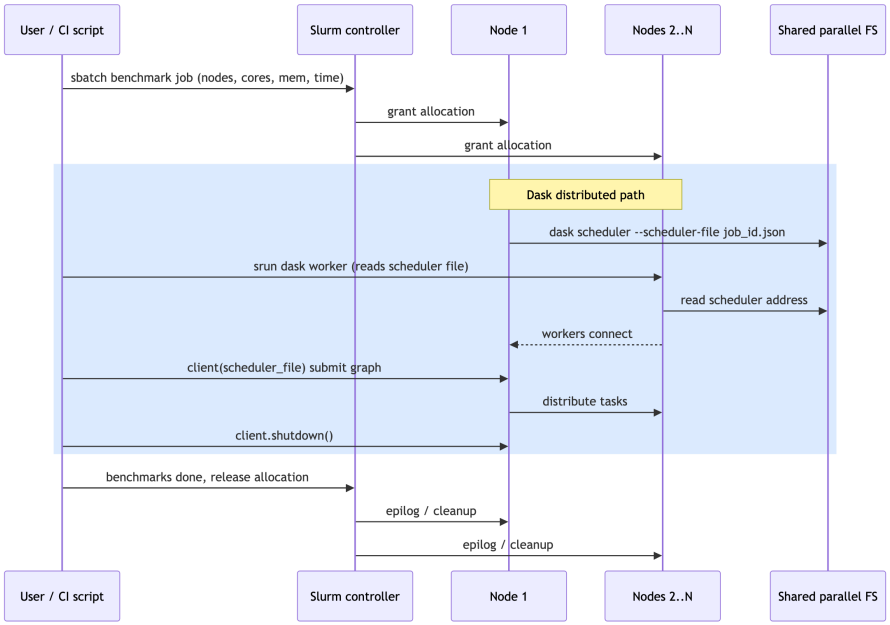
Why this matters for the measurements. All timings in Section 5 are steady-state *application* times; cluster startup and teardown are excluded and reported here once. The exclusion is deliberate: startup cost is a property of the deployment pattern, not of the framework's execution model, and folding cluster-initialisation time into a short benchmark run would distort the comparison the thesis is designed to make. The full launch scripts are part of the released artefact (Contribution 1). Figure 6 illustrates both allocation lifecycles side by side.

4.3 Benchmark Rationale: What Each Workload Tests

Each benchmark was chosen to stress a specific aspect of the framework architecture. Table 4 maps each workload to the framework component it exercises most heavily and to the mechanism that, as Section 5 will show, explains the performance outcome. The table states the hypothesised mechanism for each workload, which the evaluation then confirms with measurement.



(a) Spark standalone path: a master starts on the first node on a job-derived port, `srun` launches workers that register back, and the client polls the REST endpoint until every core is registered before submitting.



(b) Dask distributed path: the scheduler writes its address to a job-scoped file on the shared filesystem, and `srun`-launched workers discover it by reading that file rather than by polling an endpoint.

Figure 6: Allocation lifecycle for both deployment patterns inside a single Slurm job. Both share the same Slurm envelope — one `sbatch`, one allocation, one epilog — so startup and teardown cost is attributable to the deployment pattern, not the scheduler.

Table 4: Per-benchmark rationale: what each workload tests, the Dask path, and the Spark path with the mechanism behind the result.

Benchmark	**Primary stress**	**Dask path**	**Spark path and mechanism behind the result**
Data transformation	CPU-bound Python loops under the GIL	Threaded workers; the GIL serialises the 500-iteration trigonometric loop	Per-task forked Python workers; each process has its own GIL → genuine parallelism at scale
Feature engineering	Task-graph depth vs query compilation	`map_partitions` with a 100-iteration pandas loop; a shallow, wide DAG	Catalyst compiles the deep `withColumn` chain into a single fused scan pass (whole-stage code generation)
K-means (L1)	Iterative shuffle (centroid broadcast)	`dask_ml.KMeans`; k-means‖ parallel initialisation plus a task graph per iteration	MLlib `KMeans`; optimised centroid broadcast via the mature shuffle service
GMM (L2)	BLAS + iterative + memory	Custom EM: ≈12 data passes per iteration; BLAS releases the GIL → scales with threads	MLlib `GaussianMixture`: a single tree-aggregate pass per iteration; the library hides the complexity
Matrix operations	Dense BLAS vs per-row serialisation	`dask.array`: chunked NumPy; BLAS-backed Gram matrix and SVD called directly	`DistributedRowMatrix`: per-row Java↔Python serialisation before any computation begins
Linear regression	Gradient all-reduce at small scale	`dask_ml` solver; coordination overhead ≈ compute at two features	MLlib L-BFGS; similar overhead → near-tie
TF–IDF	Hashing throughput: GPU SIMT vs CPU	cuML `HashingVectorizer` on an A100; GPU single-instruction-multiple-thread parallelism	MLlib `HashingTF`: a CPU shuffle path; out-of-memory above ≈1 GB of input

4.4 Datasets

New York City Yellow Taxi Trip Records (primary dataset). Six of the seven benchmarks operate on the publicly available NYC Yellow Taxi Trip Records published by the New York City Taxi and Limousine Commission [New26], approximately 163 million rows in Parquet, reduced to approximately 142 million after standard preprocessing (removal of null records, restriction to positive trip distance and positive fare). The dataset was chosen for six reasons. *First,* its scale (on the order of 10^8 rows) is sufficient to stress distributed schedulers, partitioning, and out-of-core execution rather than fitting comfortably in a single machine's memory, which is essential for a study of distributed execution. *Second,* its mixed schema, numeric (fare, distance, tip, totals), categorical (vendor and location identifiers), and temporal (pickup and drop-off timestamps) fields, allows a single dataset to feed all six non-text workloads, so that framework differences are not confounded by differing datasets. *Third,* its columnar Parquet representation, with 64-megabyte row groups, naturally exercises partition-parallel input/output and lazy columnar reads. *Fourth,* the data are public and the experiments are therefore reproducible. *Fifth,* large-scale taxi-trip analytics on Spark has prior benchmarking precedent [Put+17; Dat21], grounding the study in a dataset that prior work has already characterised and facilitating cross-study comparison. *Sixth,* the workloads map onto genuine urban-mobility applications, namely trip segmentation, demand modelling, and fare prediction, satisfying the requirement that each workload be justified by a real use case.

Amazon Product Reviews (secondary dataset, text only). Because the taxi data contain no free-text column, and a natural-language workload requires a genuine text corpus, the text-processing benchmark uses a product-review corpus (the *CDs & Vinyl* subset of the Amazon Reviews collection [NLM19]), at sizes from approximately 701,000 to approximately 4.8 million reviews. This dataset is introduced solely for the text workload; the six tabular benchmarks remain on the shared taxi dataset, preserving the property that the data are held constant wherever that property applies.

4.5 Experimental Setup

All experiments run on the GWDG Scientific Compute Cluster (SCC), a heterogeneous, Slurm-managed HPC system. Spark is deployed via the `spark/3.5.1` environment module in standalone cluster mode; Dask is deployed via `dask.distributed`, with workers launched through `srun`. For every configuration, both frameworks' clusters are started fresh, run all benchmarks, and are torn down before the next configuration begins, eliminating state contamination between runs (Section 2.3).

Hardware and partitions. Both frameworks ran on the same `scc-cpu` partition (Intel Xeon host nodes), so the framework comparison is not confounded by a host-CPU-architecture difference. The sole exception is Dask's GPU-accelerated TF–IDF path (Section 5.8), which requires the `scc-gpu` partition's NVIDIA A100 accelerators and therefore runs on that partition's AMD EPYC host nodes instead; Spark's TF–IDF implementation remains CPU-only on `scc-cpu`. Table 5 records the configuration.

Table 5: Hardware and partitions. Cells marked ‡ are GWDG SCC/NHR public-documentation defaults.

	Dask (Python 3.11 and 3.13)	**Spark 3.5.1**
Partition	`scc-cpu` (Emmy island); `scc-gpu` (Grete island) for the TF–IDF GPU path only	`scc-cpu` (Emmy island)
Host CPU	Intel Xeon Platinum (Sapphire Rapids), 96 cores/node‡ (AMD EPYC 7xx3, Zen3, on the TF–IDF GPU nodes only)	Intel Xeon Platinum (Sapphire Rapids), 96 cores/node‡
GPU	4 × NVIDIA A100 (mixed 40 GB and 80 GB); used only by the TF–IDF workload	none
Interconnect	InfiniBand HDR (200 Gb/s)‡	InfiniBand HDR (200 Gb/s)‡
Memory / cpus-per-task swept†	64–128 GB / 32–64	64–128 GB / 32–64
Per-executor sizing	one worker per node, all cores in the allocation	standalone, four-core executors
Walltime	8 h	8 h

‡ *Values per GWDG SCC/NHR public documentation.* † *Model training additionally sweeps cores down to 16 and memory down to 32 GB (§5.7).*

Memory available to Spark. Spark's executor memory is derived formulaically from the Slurm allocation rather than left at a default. With executors sized at four cores each, the number of executors per node is the per-task core count divided by four, and each executor receives `(total node memory − 8 GB headroom) / executors_per_node`. At the principal 128-GB-per-node configurations this yields, in aggregate, on the order of 240 GB of executor memory across the two nodes (the precise per-executor figure depends on the core count); at 64-GB-per-node configurations the aggregate is correspondingly halved. The 8-GB-per-node headroom is reserved for the operating system, the worker daemon, and off-heap and shuffle buffers. This explicit sizing is the reason the text-processing out-of-memory failures of Section 5.8 are notable: they occur despite roughly 240 GB of aggregate executor memory and are therefore not a raw-RAM shortage but a consequence of the hashed-feature expansion, as analysed there.

Why a single Spark version. Dask was benchmarked under both Python 3.11 and Python 3.13 because its scheduler and its Python-level compute execute in the CPython interpreter, making the interpreter version a direct performance factor (Section 2.4). Spark was held at a single Python version, for two reasons. First, its execution engine, comprising Catalyst, Tungsten, the shuffle service, and MLlib, runs on the JVM [Dün+17; Arm+15], with the Python process acting only as a driver and a user-defined-function host; the interpreter version is therefore not on Spark's compute critical path, so varying it would not be expected to move Spark's results in the way it moves Dask's. Second, varying it would in practice have required changing the Spark engine version as

well, because Spark 3.5.1 predates official Python 3.13 support; doing so would confound the Python-version variable with a Spark-engine-version change and undermine the very comparison it was meant to inform. Holding Spark at one Python version is thus the design choice that *protects* the cleanliness of the Python-version comparison, which is conducted entirely within Dask, on fixed `scc-cpu` hardware.

Parameter sweep. The core framework comparison covers a factorial grid over node count ∈ 2, 4, per-node memory ∈ 64, 128 GB, and cores per task ∈ 32, 64, giving eight resource configurations per framework for each of the four tabular benchmarks that admit the full grid (data transformation, feature engineering, K-means, matrix operations); the Gaussian-mixture workload is restricted to the 128 GB memory level (four configurations), since it exhausts memory at ≤64 GB per node. The model-training workload uses a separate, wider sweep that additionally varies cores per task down to 16 and memory down to 32 GB (Section 5.7), which is why its configuration count differs from the rest of the suite. Tables 6 and 7 show the per-workload completion of the sweep for Dask and Spark respectively.

Table 6: Parameter-sweep completion: Dask (Python 3.13, two-node slice shown). All cells completed except GMM at ≤64 GB (out-of-memory).

Workload	C32/64G	C64/64G	C32/128G	C64/128G
Data transformation	✓	✓	✓	✓
Feature engineering	✓	✓	✓	✓
K-means	✓	✓	✓	✓
GMM (L2)	OOM	OOM	✓	✓
Matrix operations	✓	✓	✓	✓
Linear regression	✓	✓	✓	✓

Table 7: Parameter-sweep completion: Spark 3.5.1 (two-node slice shown). Same pattern: GMM is out-of-memory at ≤64 GB.

Workload	C32/64G	C64/64G	C32/128G	C64/128G
Data transformation	✓	✓	✓	✓
Feature engineering	✓	✓	✓	✓
K-means	✓	✓	✓	✓
GMM (L2)	OOM	OOM	✓	✓
Matrix operations	✓	✓	✓	✓
Linear regression	✓	✓	✓	✓

The four-node slice follows the same pattern. Across nodes ∈ 2, 4, memory ∈ 64, 128 GB, and cores ∈ 32, 64, the core comparison comprises eight configurations per

framework/interpreter variant for each of the five workloads that use this grid (GMM restricted to four, as above); model training is swept separately (Section 5.7).

Measurement protocol. Instrumentation captures wall-clock timing per named pipeline phase for both frameworks. For Dask, the benchmark harness snapshots scheduler state (`client.scheduler_info()`) before and after each run, allowing the cumulative per-operation compute time and inter-worker transfer volume to be derived; this is the source of the "effective concurrency" figures used in Section 5.2. For Spark, the harness queries the REST API (`/api/v1/applications/id/stages` and `/executors`) for per-stage executor time, garbage-collection time, shuffle read and write bytes, and spill metrics. Following the JVM-benchmarking guidance of Georges et al. [GBE07], the model-training workload is run five times per configuration with the cold first run reported separately and the steady-state mean and dispersion reported alongside; this protocol is what allows Section 5.7 to distinguish a genuine steady-state difference from a warm-up artefact.

4.6 Benchmark Implementations

Each benchmark is implemented in Python with strict semantic parity between the two frameworks except where a divergence is explicitly documented. The implementations are described below in terms of what each computes, read directly from the source.

Data transformation and aggregation. The workload reads ten taxi columns from Parquet, derives a trip-duration field and filters to valid trips, derives average speed and temporal fields, applies a deliberately heavy 500-iteration vectorised sine–cosine kernel per partition (via `map_partitions` on Dask and a `pandas_udf` on Spark) to produce a complexity metric, performs a group-by aggregation keyed on vendor, and selects the hundred longest trips. It is the canonical big-data ETL and online-analytical-processing workload and deliberately combines three bottleneck classes, namely columnar input/output at load, a CPU-bound numeric kernel, and a shuffle at the aggregation, exercising the DataFrame and SQL execution path where the two engines differ most for tabular analytics [Hua+10; Wan+14]. Because the two implementations are mechanically symmetric, the workload constitutes a clean test of the parallelism model (Section 5.2).

Feature engineering. The workload reads five numeric columns, constructs several ratio and composite features, then applies a 100-iteration loop of additive and multiplicative transforms, expressed as a `map_partitions` loop on Dask and as 100 chained `withColumn` expressions on Spark, before computing a final score. It isolates iterative, expression-heavy feature construction with negligible shuffle, and is chosen specifically to expose how each engine compiles a long chain of column expressions: Catalyst can fuse such a chain into a single physical pass through whole-stage code generation [Arm+15; Neu11], whereas Dask materialises a longer blocked task graph. The implementation asymmetry is a lazy column chain on Spark against an imperative loop on Dask.

K-means clustering (Level 1). The workload loads six feature columns, removes nulls, filters to positive distance and fare, standardises the features, and fits K-means with ten clusters and a fifty-iteration cap (`dask_ml.cluster.KMeans` on Dask, MLlib `KMeans` on Spark). K-means is the canonical unsupervised micro-benchmark: distance-based, iterative (Lloyd's algorithm), and characterised by a per-iteration centroid all-reduce and broadcast that stresses inter-worker communication. Its distributed initialisation is itself non-trivial: the default parallel seeding scheme, k-means‖ [Bah+12] (the distributed form of the k-means++ squared-distance seeding of Arthur and Vassilvitskii [AV07]), makes several oversampling passes over the data before Lloyd iterations begin, an overhead that

turns out to dominate the Dask result (Section 5.4).

Gaussian mixture model / expectation–maximisation (Level 2). The workload applies the same load, preprocess, and scale stages and then fits a Gaussian mixture model with ten components and a tolerance of 10^{-3}. Spark uses MLlib `GaussianMixture`; Dask uses a hand-written expectation–maximisation routine with Cholesky-factored precision matrices, squared-Mahalanobis log-Gaussian densities, a responsibility matrix persisted across workers, and per-component scatter matrices, together with numerical safeguards (a jitter fallback for the Cholesky factorisation, covariance regularisation, and log-sum-exp stabilisation of the responsibilities). This is the harder clustering variant requested during supervision: it escalates algorithmic intensity at the same data scale, performing dense per-component linear algebra each iteration and materialising a large responsibility matrix (≈11 GB at 142 M rows × 10 components), rendering it simultaneously compute-, communication-, and memory-bound and requiring at least 128 GB per node. Together with K-means it forms the Level-1-to-Level-2 escalation pair. The library-versus-custom divergence is a sanctioned methodological choice (Section 6.4).

Matrix operations. The workload loads the six filtered taxi feature columns into a tall-and-skinny dense matrix (≈142 M × 6) and computes the Gram matrix ($A^{\mathsf{T}}A$), a truncated singular-value decomposition (low rank), column means and variances, and an element-wise reduction. Dask uses `dask.array` with BLAS-backed kernels and a randomised compressed singular-value decomposition [HMT11]; Spark uses the row-oriented distributed `RowMatrix`. The workload tests the distributed dense-linear-algebra path that underlies principal-component analysis, covariance estimation, and least-squares fitting, and deliberately probes a known design boundary: Dask extends NumPy and BLAS to distributed arrays, whereas Spark's distributed matrices are row-oriented and not designed for dense BLAS-style operations [Zad+16; TK18; Gu+15]. Both implementations materialise fully, Dask through persist-and-wait and Spark through cache-and-count, so the comparison is fair rather than an artefact of lazy evaluation.

Model training: linear regression. The workload reads four columns, preprocesses them, sets the design matrix to trip distance and passenger count and the target to fare amount, and trains a linear regression model (Dask via a gradient-based solver, Spark via MLlib `LinearRegression` with the limited-memory BFGS optimiser; [LN89]) five times, reporting the mean and a 95% confidence interval. It is the canonical supervised-learning benchmark: an iterative convex optimisation whose per-iteration gradient all-reduce stresses the communication layer differently from the centroid exchange of K-means [Men+16]. The two-feature problem is deliberately small, which, as Section 5.7 shows, makes per-iteration coordination overhead, rather than arithmetic, the dominant cost.

Text processing: TF–IDF. The workload reads a review-text column, normalises it, applies a hashing vectoriser, and then applies an inverse-document-frequency weighting. The Dask implementation runs on an NVIDIA A100 GPU (the cuML hashing vectoriser and TF–IDF transformer over cuDF and CuPy; [Oku+17]), whereas the Spark implementation runs on the CPU (a tokeniser, a hashing term-frequency stage with 2^{20} features, and an inverse-document-frequency stage in MLlib). The workload adds the natural-language branch absent from the six tabular workloads. It is included specifically to benchmark the GPU-accelerated path against Spark's CPU pipeline; the comparison is intentional and ecosystem-level rather than a like-for-like hardware test (Section 6.4). The benchmark implementations are adapted from NVIDIA's `rapids-examples` repository, used as a code basis rather than cited as a source.

4.7 Implementation-Complexity Evaluation Methodology

Implementation complexity is treated as a first-class research concern, answering **RQ4** through three complementary methods applied to each of the twelve benchmark scripts (six workloads × two frameworks; the GMM pair adds the custom-versus-library contrast).

Static code metrics. Source lines of code (SLOC) and logical lines of code (LLOC), average cyclomatic complexity per function, and the maintainability index are measured with the `radon` tool (version 6.0.1) on all scripts. Cyclomatic complexity is reported on radon's letter scale (A = 1–5, B = 6–10), and the maintainability index on its 0–100 scale (higher is better). These metrics capture the size and branching structure of the *user-facing* code, the code a practitioner must write and maintain, as distinct from the framework internals.

Structured qualitative assessment. Alongside the static metrics, each Dask/Spark implementation pair is assessed along four fixed dimensions: conceptual difficulty, API familiarity (proximity to pandas and NumPy idioms), debugging surface, and migration cost. The dimensions are fixed in advance to keep the assessment consistent across pairs. What is reported (Section 5.12) is the implementer's qualitative judgement per dimension, grounded in having written both sides of every pair, rather than a numeric score tabulated per implementation.

Impedance-mismatch catalogue. Qualitative friction points observed during implementation are catalogued systematically: the structural mismatches that force one framework's code to be longer or more intricate than the other's for the same task. The catalogue is presented in Section 5.12 and is the qualitative complement to the static metrics.

5 Evaluation

This section reports and interprets the measured results as a single evaluation combining measurement, mechanism, and caveats per workload, with the discussion of broader implications, limitations, and threats deferred to Section 6. The section is organised to answer the four research questions of Section 4.1 in turn. Sections 5.2–5.9 work through the individual workloads and the Python-version axis, supplying the per-workload evidence for **RQ1** (relative performance) and **RQ3** (architectural causes); Section 5.10 aggregates the scaling behaviour for **RQ2**; Section 5.11 synthesises the recurring architectural mechanisms for **RQ3**; and Section 5.12 reports the implementation-complexity findings for **RQ4**. Each subsection opens with a single claim; read in sequence, those opening claims tell the whole story: *no framework dominates unconditionally, the winner is set by which resource saturates first, and the software environment is itself a first-class variable.*

Unless otherwise stated, all tabular workloads use the full 142-million-row taxi dataset across the eight hardware configurations of Section 4.5, and all Dask figures in the cross-framework comparison refer to the Python 3.13 build so that the framework contrast is not confounded with the interpreter contrast (which is isolated separately in Section 5.9).

5.1 Overview of Measured Outcomes

Verdict. The benchmark suite produces a workload-dependent verdict: Apache Spark wins four of the six tabular workloads (by factors of 1.7 to 2.7); Dask wins dense linear algebra by more than an order of magnitude (about 17× on average) and GPU-accelerated text processing by roughly tenfold; and the two frameworks are in statistical parity on linear regression, a distribution of outcomes that is itself the primary finding, because no framework dominates unconditionally.

The headline numbers are collected in Table 8 and visualised, on a logarithmic axis, in Figure 7. The pattern is not a narrow points-decision in which one framework edges ahead everywhere; it is a genuine split in which the margin and its direction both depend on the workload. Spark's advantages cluster on the four workloads whose inner loop is a CPU-bound Python computation or a shuffle-bearing aggregation (data transformation, feature engineering, K-means, and the Gaussian mixture), where its process-based executors and its Catalyst-compiled query plans convert hardware into parallel throughput. Dask's advantages cluster where the inner loop is dense numerical linear algebra that releases the interpreter lock into BLAS, namely matrix operations, and where a GPU path is available for hashing-bound text feature extraction. On linear regression, a deliberately small convex problem, the two are indistinguishable. The winner-by-configuration view in Figure 8 confirms that these verdicts are stable across the eight configurations rather than artefacts of any single allocation: the blue/red boundary barely moves as cores, memory, and node count vary.

Table 8: Measured wall-clock times (seconds) at best and worst configuration; Spark and Dask 3.13 shown. All tabular benchmarks use the full 142 M-row taxi dataset across eight hardware configurations. Best/worst are taken across all measured configurations.

Benchmark	Spark best	Dask best	Spark worst	Dask worst	Winner
Data transf.	134 s	236 s	255 s	257 s	Spark at scale
Feature engr.	7 s	19 s	27 s	51 s	**Spark ∼2.7×**
K-means (L1)	49 s	83 s	82 s	114 s	**Spark ∼1.7×**
GMM (L2)	178 s	416 s	339 s	1034 s	**Spark ∼2.3×**
Matrix ops	68 s	6 s	167 s	8 s	**Dask ∼17×**
Lin. regression	3 s	2.3 s	5 s	3.5 s	Near-tie
TF–IDF[†]	123 s	12 s	123 s	21 s	**Dask GPU ∼10×**

[†] *TF–IDF from single-config runs (2 nodes, 128 GB, 32 CPUs + 1 A100 GPU for Dask; CPU-only for Spark, ∼15 GB per executor / ∼240 GB aggregate at this configuration).*

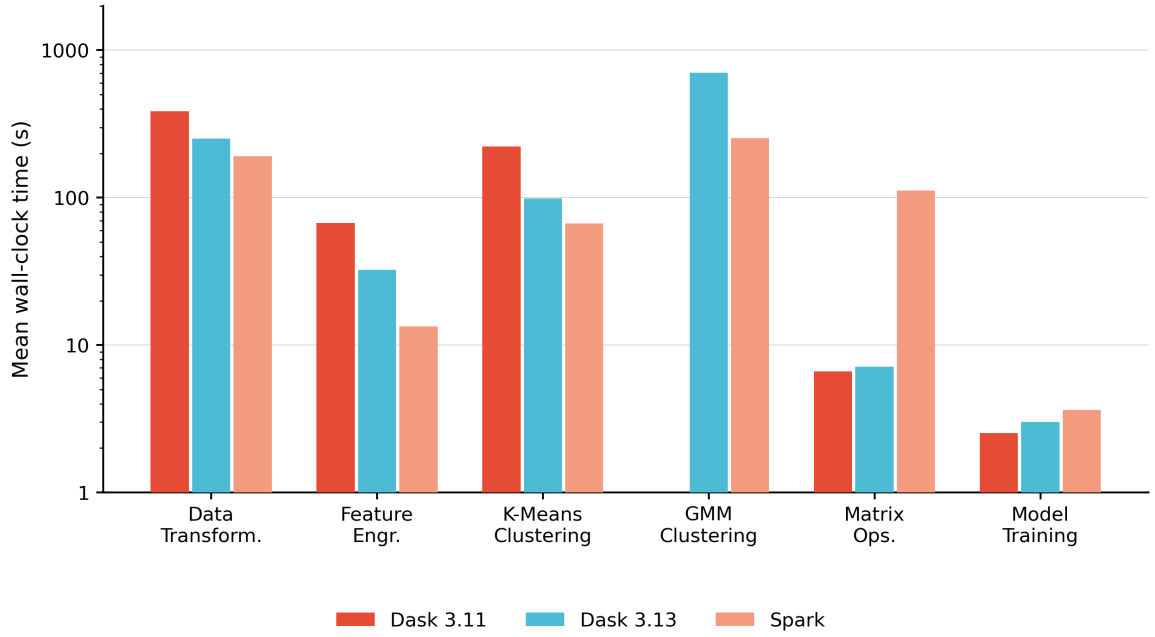


Figure 7: Overall benchmark comparison (mean wall-clock time across all eight configurations, log scale). GMM has no Dask 3.11 bar as that workload was not benchmarked under Python 3.11. TF–IDF is omitted from this figure entirely: unlike the other six workloads, it was run at a single fixed configuration only (Table 8, footnote), not swept across configurations, so it has no cross-configuration mean to plot. Spark leads on the four CPU-/shuffle-bound tabular workloads.

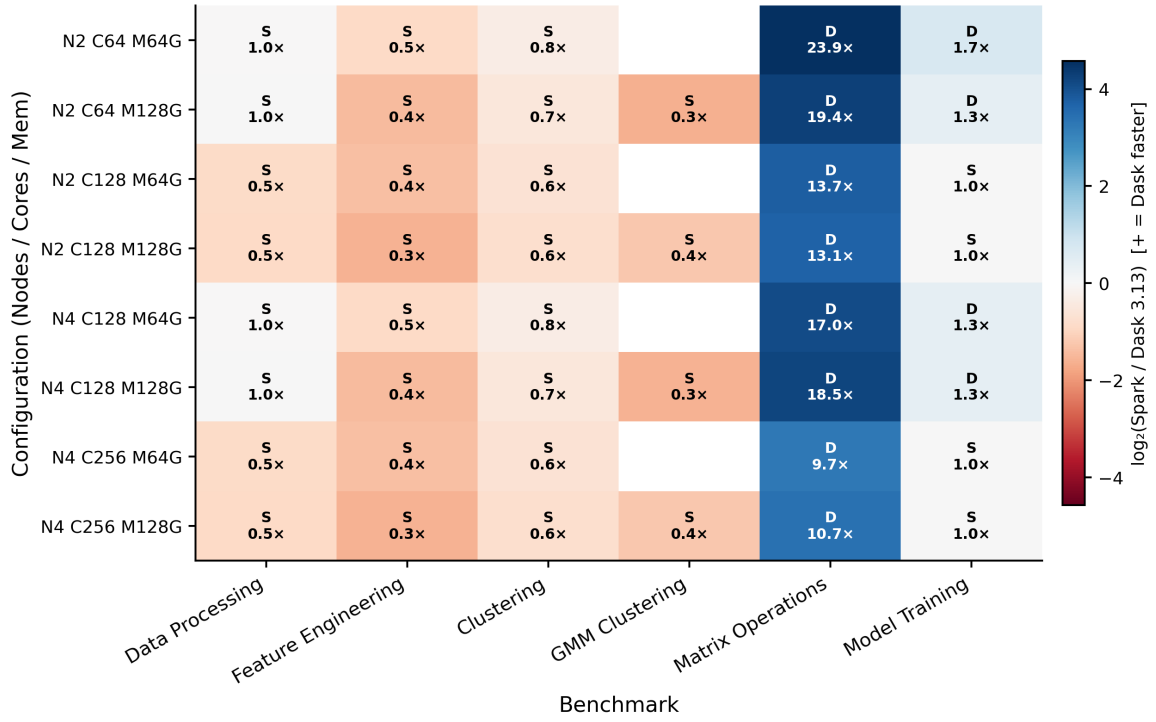


Figure 8: Winner heatmap, Dask 3.13 versus Spark across all configurations. Blue indicates Dask faster, red indicates Spark faster; blank cells indicate configurations that could not complete due to out-of-memory errors (Gaussian mixture at $\leq$64 GB per node for both frameworks). The verdict is stable: Spark wins the four tabular workloads, Dask wins matrix operations decisively.

The remainder of the section explains *why* each verdict falls as it does, because the explanation, and not the leaderboard, is what transfers to workloads beyond this suite.

5.2 Data Transformation and Aggregation

Verdict. Spark processes the 142-million-row extract–transform–load pipeline about 1.2× faster than Dask on average (mean 217 s versus 256 s), and the gap widens to 1.9× at 64 cores, because Spark's per-task Python processes achieve genuine lock-free parallelism as the core count doubles, whereas Dask's threads plateau at an effective concurrency bounded by the Global Interpreter Lock.

This workload is the cleanest test of the parallelism model in the entire suite, because the two implementations are mechanically symmetric: the same ten-column Parquet read, the same derived fields, the same deliberately heavy 500-iteration vectorised sine–cosine kernel applied per partition, and the same vendor-keyed aggregation, expressed once through `map_partitions` on Dask and once through a `pandas_udf` on Spark. The run is dominated by the CPU-bound kernel; the aggregation shuffles almost nothing (Spark reports on the order of 0.2 megabytes), so the result reflects how each engine parallelises a Python-level numeric loop and almost nothing else.

The two engines diverge exactly as their intra-node execution models predict. Spark improves from 254 seconds at 32 cores to 136 seconds at 64, a near-linear response to additional cores, because each task runs in its own forked Python worker and therefore executes the kernel free of any shared interpreter lock. Dask remains essentially flat at about 255

seconds, and, decisively, does not improve when the thread count per worker doubles from 32 to 64. The scheduler instrumentation explains why: it reports approximately 1688 seconds of cumulative per-operation compute completing in approximately 255 seconds of wall-clock time, an *effective concurrency* of only about 6.6, far below the available thread count (Appendix A.3). Adding threads cannot help because the Python-level body of the kernel serialises on a single Global Interpreter Lock, while the per-iteration churn of temporary NumPy arrays saturates memory bandwidth, a memory-bound regime in the sense of the roofline model [WWP09]. The phase breakdown in Figure 9 confirms that the compute phase, not input/output, dominates for both Dask builds: the Parquet read accounts for only about 10 seconds of the roughly 255-second run. Input/output is thus never the binding constraint on this workload, despite being one of the three bottleneck classes the workload was designed to exercise (Section 4.3); the CPU-bound kernel and the GIL-limited thread concurrency dominate instead, which is itself informative about where a practitioner should expect the bottleneck to sit on a similarly columnar, memory-resident dataset. Spark is not shown in that figure because its own instrumentation does not produce a comparable wall-clock decomposition (see the note following the figure), but its wall-clock total is reported above and in Table 8, and its cumulative executor time and shuffle volume for this and the other tabular workloads are given in Appendix Figure 21.

This behaviour is consistent with the design of Dask's threaded scheduler [Roc15], with the scaling gaps observed when Dask and Spark are compared on HPC systems [Dug+23], and with the per-task scheduler-overhead characterisation of Böhm and Beránek [BB20].

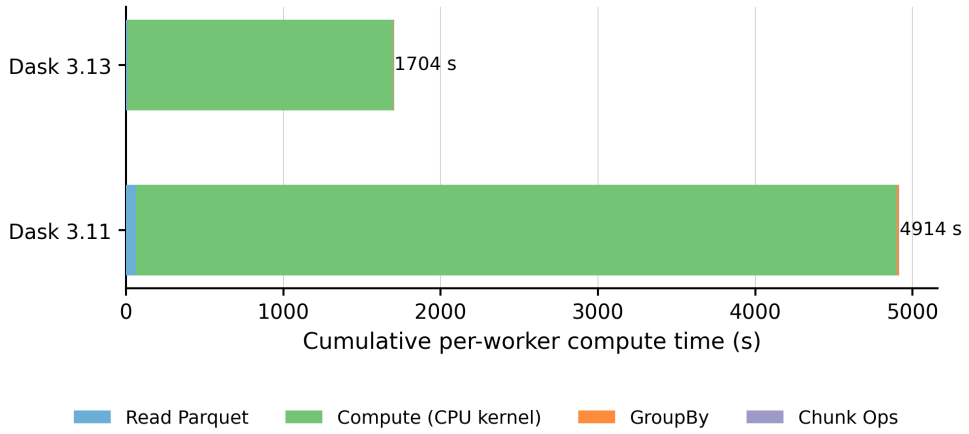


Figure 9: Data-processing phase breakdown, Dask 3.11 and 3.13. The compute (CPU) phase dominates for both versions; input/output is a minor component. The x-axis is cumulative per-worker compute time, not wall-clock; Dask's wall-clock totals for this workload are 386 s (3.11) and 256 s (3.13), per Table 10. Spark is omitted: its stage instrumentation is collected through the Spark REST API's `executorRunTime` field, which reports cumulative task time summed across all executors for a stage rather than wall-clock duration, and which decomposes the job into different stages (a count, a vendor-keyed aggregation, and a top-trips ranking step) than the four phases timed on the Dask side. The two measurements are therefore not on a common footing and are not shown together; Spark's wall-clock total for this workload is reported in Table 8.

5.3 Feature Engineering

Verdict. Spark completes the 100-iteration feature-construction pipeline about 2.7× faster than Dask at the best configuration (7 s versus 19 s), because Spark's Catalyst optimiser compiles the deep `withColumn` chain into a single tight pass over the data while Dask executes the equivalent as an interpreted per-partition pandas loop.

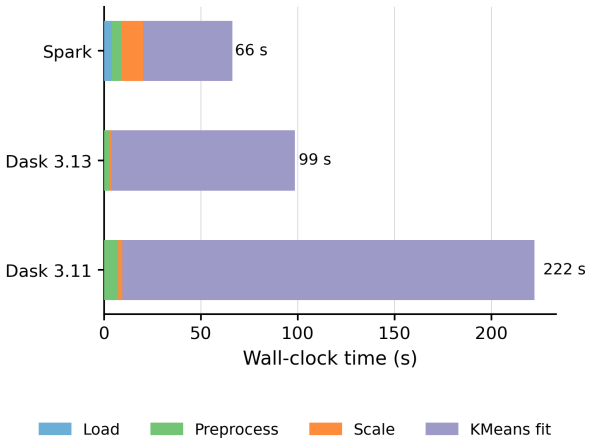
The workload chains many dependent column transformations, a five-feature base followed by a 100-iteration refinement of a composite feature, and the two frameworks express that chain very differently. On Spark the 100 iterations are built as a lazy chain of `withColumn` expressions; Catalyst optimises the resulting deep expression tree and, through whole-stage code generation, compiles it into a single tight pass over the data, so that the loop effectively disappears at planning time [Arm+15; Neu11]. On Dask the same 100 iterations execute as a literal pandas loop inside `map_partitions`, materialising every intermediate Series at run time. The practical implication is real, because the idiomatic way to write this pipeline in each framework is exactly what was written, so the measured gap is the gap a practitioner would actually encounter.

5.4 K-means Clustering (Level 1)

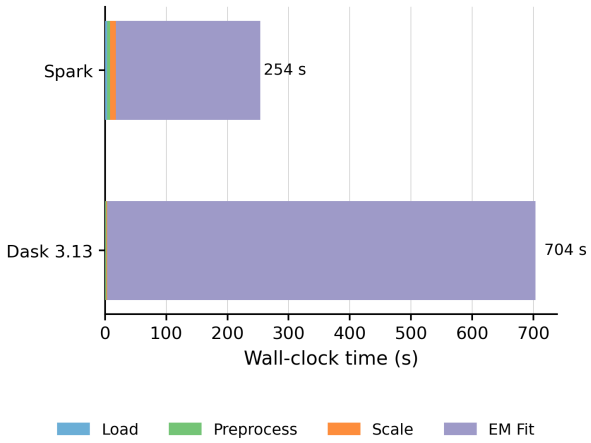
Verdict. Spark fits K-means faster than Dask 3.13 across all configurations (49–82 s versus 83–114 s, about 1.7× at the best configuration), a gap attributable to k-means‖ parallel-initialisation overhead and Dask's task-graph coordination cost rather than to arithmetic work (Dask converges in only three Lloyd iterations).

A single control fact settles the interpretation. The Dask implementation converged in only three Lloyd iterations in every configuration, yet still required roughly 96 seconds. With three passes over six-dimensional data the arithmetic is trivial, so the elapsed time cannot be floating-point work; it is overhead. Two sources account for it: the default parallel seeding scheme, k-means‖ [Bah+12] (the distributed form of the k-means++ squared-distance seeding of Arthur and Vassilvitskii [AV07]), which makes several over-sampling passes over the full dataset before Lloyd iterations even begin, and the per-iteration construction and coordination of the Dask task graph characterised by Böhm and Beránek [BB20]. Spark's MLlib K-means, by contrast, is a mature implementation in which each iteration is a single tree-aggregate pass that scales with the core count, improving from 55 seconds at 32 cores to 40 at 64 [Men+16]. The phase breakdown in Figure 10a shows the fit phase dominating all engines and locates the Python-version improvement (Dask 3.11's 202.8 seconds falling to Dask 3.13's 96.3) without closing the gap to Spark. The gap is therefore one of implementation maturity and fixed initialisation overhead rather than algorithmic arithmetic, consistent with the K-means characterisations of Huang et al. [Hua+10] and Harifi et al. [HBK17].

A secondary impression from these runs is noted here only to flag it for future work, not to assert it as a finding: the benchmark suite ran each configuration once, and the resulting Dask cluster assignments appeared visually less variable across configurations than Spark's. No repeated-run variance statistics were collected for clustering, so this thesis makes no consistency claim; Section 6.5 specifies the re-run that would test this directly. The Level 1 verdict is therefore stated on speed alone: Spark is faster. Figure 10 places the K-means and GMM phase breakdowns side by side for comparison.



(a) Mean across all eight measured configurations. K-means: the fit phase dominates all engines; Dask 3.13 (99 s) improves over Dask 3.11 (222 s) but remains slower than Spark (66 s).



(b) Mean across the four measured configurations (128 GB per node only; GMM exhausts memory at 64 GB). Gaussian mixture / expectation–maximisation: Spark's tree-aggregate (254 s) is about 2.8× faster than the Dask custom routine (704 s mean).

Figure 10: Clustering phase breakdown.

5.5 Gaussian Mixture Model / Expectation–Maximisation (Level 2)

Verdict. Spark fits the Gaussian mixture 2.3 to 3.1× faster than the hand-written Dask expectation–maximisation routine (mean 254 s versus 704 s), because Spark's MLlib accumulates all ten components' sufficient statistics in a single tree-aggregate pass per iteration while the Dask routine performs roughly twelve full data passes; the Dask implementation, however, offers numerical robustness that the library Spark path does not expose, and both frameworks exhaust memory at 64 GB or less per node.

This is the deliberate Level-1-to-Level-2 escalation: the same data scale, a markedly heavier algorithm. The decisive evidence mirrors the K-means analysis but points the other way on cause. The Dask routine actually ran *fewer* iterations, having already converged, yet was about 2.5 times slower, so the difference is per-iteration cost, not iteration count. The mechanism is structural. The Dask routine persists the dense responsibility matrix (approximately 11 gigabytes for 142 million rows and ten components) at each iteration, then loops over the ten components, each performing a full-data weighted covariance pass with its own persist-and-wait synchronisation, amounting to roughly a dozen full passes over the data per iteration. Spark's MLlib `GaussianMixture.fit` instead accumulates the sufficient statistics for all ten components in a single tree-aggregate pass per iteration [Men+16]. Twelve passes against one is the whole story of the 2.5× gap, as Figure 10b summarises.

Two riders complete the picture. First, the Dask routine does scale with thread count (from roughly 1030 seconds at 32 cores to 491 at 64), precisely because its maximisation step is dominated by BLAS matrix multiplications that release the interpreter lock, the same mechanism that wins Dask the matrix workload (Section 5.6). Second, the speed comparison understates what the Dask code buys in return: it carries explicit numerical safeguards (Cholesky-factored precision matrices with a jitter fallback, covariance

regularisation, and log-sum-exp stabilisation of the responsibilities) that the MLlib path does not surface to the user. The richer finding is therefore a tractable trade-off between framework-native efficiency, exemplified by Spark, and implementation flexibility and numerical control, exemplified by Dask. Both frameworks require at least 128 gigabytes per node for this workload and fail with out-of-memory errors at 64 gigabytes or below, which is itself a reportable parity result: the responsibility matrix, not the framework, sets the memory floor. The ≈11 GB responsibility matrix is not the only large structure: the Dask implementation also keeps the raw and scaled feature arrays persisted for the fit's lifetime and, each iteration, materialises a comparably sized weighted-feature copy per component before freeing it, so several such structures coexist rather than one; no peak-RSS instrumentation was collected (Section 6.3) to attribute the remaining gap to 64 GB more precisely. Spark's failure at the same threshold has no comparable custom structure to inspect, since `GaussianMixture.fit` is a single library call, and more plausibly reflects JVM heap fragmentation and garbage-collection overhead against the executor budget of Appendix B.2. The relevance of expectation–maximisation as a heavier distributed-clustering workload is established by Xiao et al. [Xia+20] and Esmaeilpour et al. [Esm+16].

5.6 Matrix Operations

Verdict. Dask completes the full matrix workload in 6 to 8 seconds while Spark requires 68 to 167 (the largest margin in the suite, about 17× on the mean), because Dask's chunked `dask.array` calls BLAS-backed kernels directly, whereas Spark must serialise every row across a per-row Java-to-Python boundary to construct the `RowMatrix` before any computation can begin.

The workload is a tall-and-skinny dense matrix of approximately 142 million rows by six columns, on which it computes the Gram matrix, a truncated singular-value decomposition, column statistics, and an element-wise reduction. The two engines sit on opposite sides of a known design boundary. Dask represents the matrix as a chunked `dask.array` and executes native BLAS: the Gram matrix is a blocked multiplication in which each chunk yields a six-by-six partial that is tree-reduced, and the compressed singular-value decomposition is a randomised algorithm [HMT11] that operates on a small sketch rather than the full matrix. These BLAS-backed kernels release the Global Interpreter Lock and parallelise fully across the worker's threads [Har+20]; the result is single-digit seconds, and the phase breakdown in Figure 11 shows every individual phase (load, Gram, decomposition, column statistics, reduction) completing in well under a second apart from the load.

Spark's cost lies almost entirely before the algebra starts. Constructing a `RowMatrix` requires a per-row Python lambda over approximately 142 million rows, forcing every row across the Java-to-Python-to-Java boundary with pickle serialisation before the Gram matrix or the decomposition can be computed. This materialisation and row-wise serialisation, not the trivial six-by-six arithmetic, dominate the 68-to-167-second runtime; in Figure 11 each individual Spark phase exceeds the entire Dask total. This is the documented weakness of MapReduce- and Spark-style systems for dense linear algebra relative to array-native systems [TK18], the row-matrix cost discussed by Zadeh et al. [Zad+16], and the very motivation for purpose-built Spark matrix layers such as Marlin [Gu+15]. Crucially, both implementations materialise fully, Dask through persist-and-wait and Spark through cache-and-count, so the margin is a real difference in the cost of getting data into a com-

putable dense form, not an artefact of lazy evaluation. This workload, more than any other, is where the thread-based model that loses Section 5.2 wins decisively, and Section 5.11 develops that inversion as a central architectural finding.

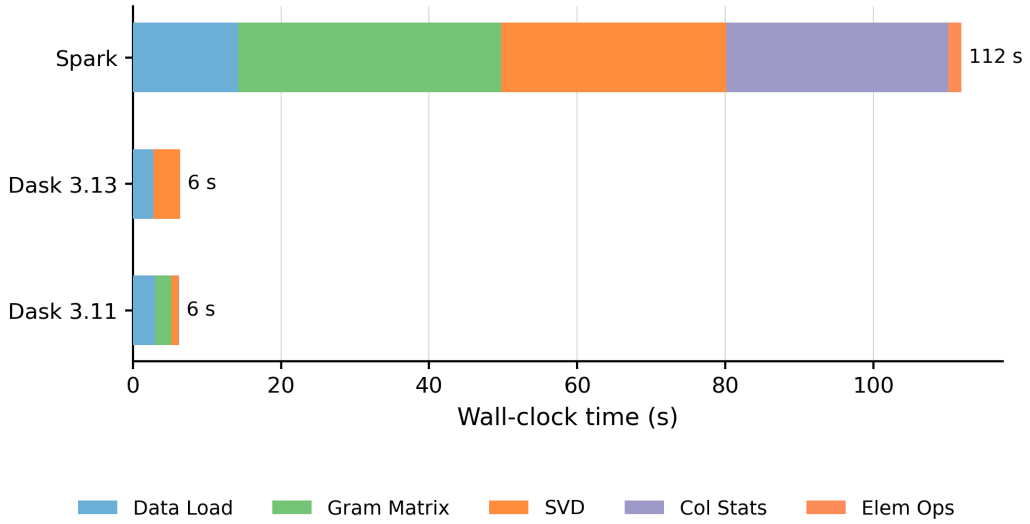


Figure 11: Matrix-operations phase breakdown. Every individual Spark phase (Gram, decomposition, column statistics) exceeds the entire Dask total of about 6 s. The bottleneck is per-row serialisation for `RowMatrix` construction.

5.7 Model Training: Linear Regression

Verdict. Dask and Spark are within scheduling noise on the five-repeat linear-regression workload (mean across all measured configurations: Dask 3.2 s, Spark 3.9 s), making this the only workload where neither framework holds a detectable architectural advantage at this problem scale.

The problem is deliberately tiny: two coefficients fitted by a limited-memory BFGS optimiser [LN89] for up to fifty iterations, repeated five times with the data pre-cached and only the fit timed. With two coefficients each gradient is almost free, so the run is dominated not by arithmetic but by the per-iteration scheduling overhead of a job launched roughly fifty times. Both frameworks show the same warm-up signature: Spark's mean is inflated by a cold first run (about 6.1 seconds, then roughly 3.6), and Dask exhibits the same profile (about 4.4 seconds, then roughly 2.4). The five-run box plot in Figure 12 makes the convergence visible: median training time overlaps closely across all three runtimes, and the two Dask interpreter builds coincide at about 3 seconds; the mean (dashed line) sits above the median in each group because the cold first run pulls it upward. At this single configuration the mean is ∼2.8 s for Dask and ∼4.2 s for Spark; the 3.2 s/3.9 s figures in the verdict above are the mean across the full model-training sweep, which additionally varies cores down to 16 and memory down to 32 GB (§4.5), while Table 14 in the appendix lists the individual two-node-configuration means these single-configuration numbers are drawn from. Following the JVM-benchmarking guidance of Georges et al. [GBE07], the cold run is reported separately from the steady-state mean precisely so that this warm-up does not masquerade as an architectural effect. The steady-state difference is a few hundred milliseconds on a roughly three-second job, which lies within scheduling variance rather than constituting a robust advantage. The

honest reading is near-parity, with Dask's lighter per-iteration coordination affording it a slight edge on a model of this size [Men+16; Dug+23].

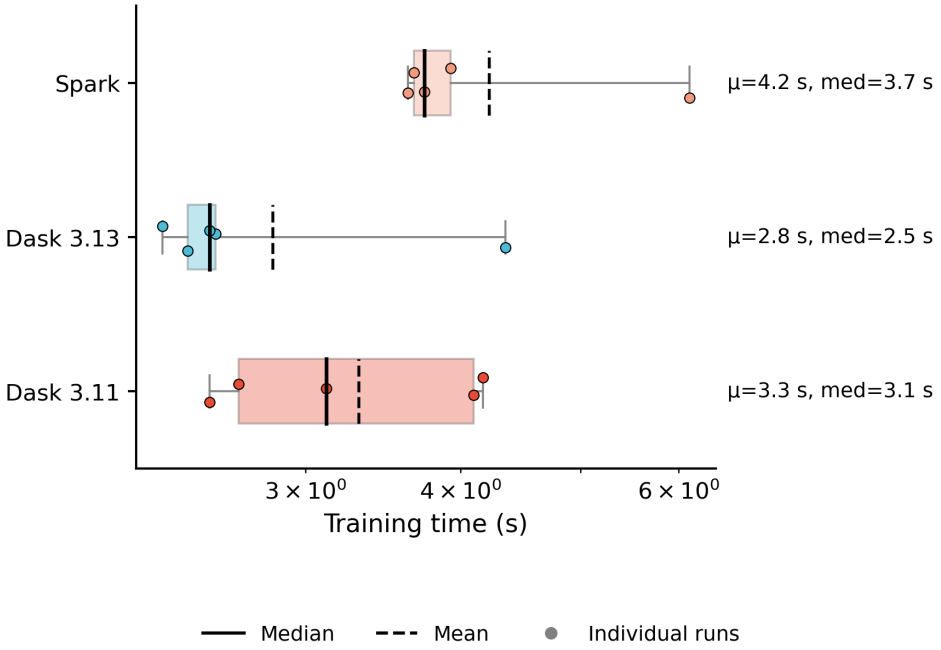


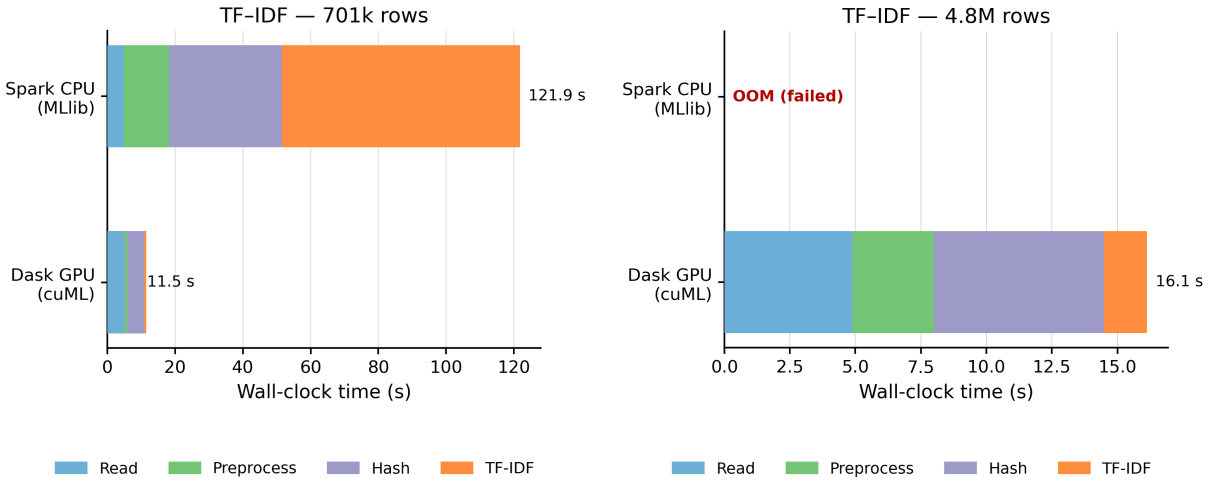
Figure 12: Linear regression, all five repeated runs at a single configuration (2 nodes, 128 GB, 32 cores per node), shown as box plots with median (solid) and mean (dashed) marked; individual runs are overlaid as points. Whiskers span the full range of the five runs. Dask 3.11 and Dask 3.13 overlap at about 3 s median. Spark's mean here is ∼4.2 s; cf. Appendix Table 14 and the 3.9 s figure quoted in the text above, which average over all measured configurations.

5.8 Text Processing: TF–IDF

Verdict. Dask's GPU-accelerated TF–IDF pipeline (cuML on an NVIDIA A100) processes the 701-thousand-row corpus about 10× faster than Spark's CPU MLlib pipeline (about 12 s versus about 123 s), and the robustness gap is even more striking: Spark fails with out-of-memory errors on corpora larger than roughly 1 GB, while Dask completes corpora up to 10 GB in a consistent ∼20 s.

This is the one deliberately cross-architectural workload in the suite, included to benchmark the GPU-accelerated path that is idiomatic for Dask against Spark's CPU pipeline, and the comparison is ecosystem-level rather than like-for-like (a framing made explicit again in Section 6). TF–IDF is embarrassingly parallel, independent per-document hashing followed by a term-frequency reduction, and maps almost perfectly onto the thousands of single-instruction-multiple-thread lanes of a GPU, so an order-of-magnitude speed advantage is expected on theoretical grounds and confirmed by the peer-reviewed GPU-TF–IDF literature [Err+15]. The measured tenfold gap at 701 thousand rows realises that expectation, and the per-phase view in Figure 13 shows why: the hashing and inverse-document-frequency phases that dominate the Spark pipeline collapse to near-zero on the GPU. At a larger 4.8-million-row corpus the same pattern holds and sharpens: Dask completes in roughly 16 seconds, while Spark fails with an out-of-memory error and never produces a completed run at this scale.

The robustness result in Table 9 is arguably more important than the speed result, because it is a difference in *kind* rather than degree. It is driven by the `HashingTF` stage's 2^{20} feature dimension and the per-partition and shuffle expansion of the hashed term-frequency vectors, which is the binding constraint rather than a raw-RAM shortage. For large-scale text feature extraction in this configuration, the CPU shuffle-and-hashing path is therefore the limiting factor, and GPU acceleration delivers both a speed and a reliability advantage. The Dask implementation is adapted from NVIDIA's `rapids-examples` repository, used as a code basis rather than cited as a source.



(a) 701k rows: both engines complete, with the Spark pipeline already dominated by its hashing and TF–IDF phases.

(b) 4.8M rows: Spark fails with an out-of-memory error at this scale; Dask completes in 16.1 s.

Figure 13: TF–IDF processing by phase: Dask GPU (cuML) versus Spark CPU (MLlib). The hashing and TF–IDF phases that dominate the Spark pipeline collapse to near-zero on the GPU.

Table 9: TF–IDF wall-clock time and failure status, Spark CPU (MLlib) versus Dask GPU (cuML), across three corpus sizes.

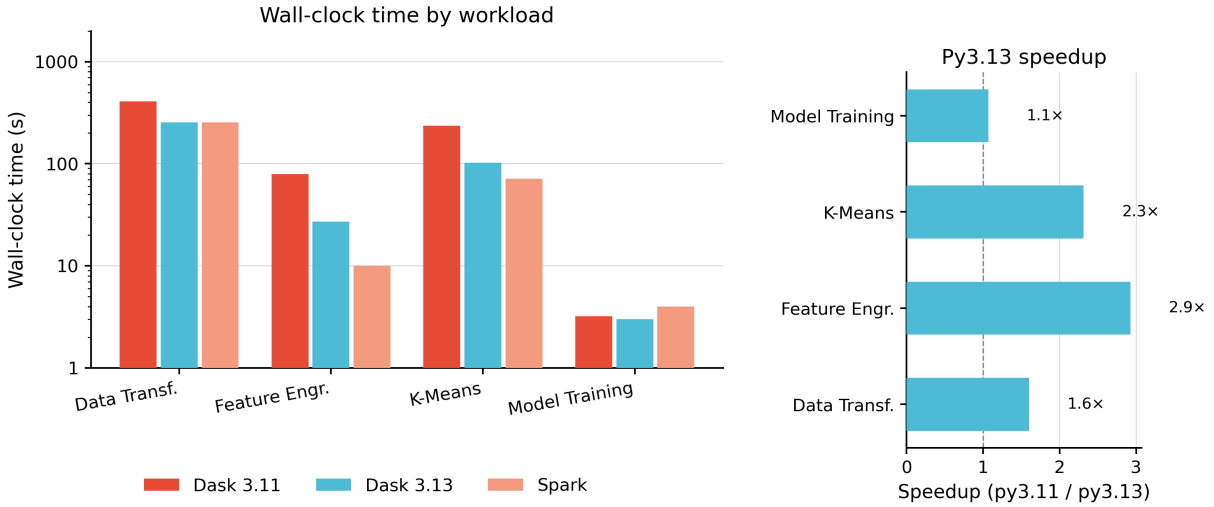
Corpus size	Spark CPU (MLlib)	Dask GPU (cuML)
~200 MB (701k rows)	~122 s	~12 s
> 1 GB	**OOM (failed)**	~20 s
Up to 10 GB	**OOM (failed)**	~20 s

5.9 Python 3.11 versus 3.13: Effect on Dask Performance

Verdict. Upgrading Dask from Python 3.11 to 3.13 on identical hardware improves wall-clock time by 1.5× on data transformation, 2.1× on feature engineering, and 2.1× on clustering, while leaving model training unchanged (about 3 s on each).

All Dask 3.13 results reported here use the standard, GIL-enabled build; the free-threaded build (PEP 703) was not evaluated.

This axis is reported separately from the framework comparison because it is the cleanest controlled experiment in the study: only the Python minor version changes, with the framework, the hardware (`scc-cpu`), the dataset, and the workloads all held fixed. The measured improvements are collected in Table 10 and plotted in Figure 14. The gains are real and concentrated on the workloads whose bottleneck is Python-level execution: data transformation (1.5×), feature engineering (2.1×), and K-means (2.1×), which is consistent with the interpreter-level improvements of the faster-CPython programme, the adaptive specialising interpreter of Shannon [Sha21] among them. Workloads that are not Python-bound show no benefit: matrix operations, dominated by BLAS, is unchanged at about 7 seconds, and model training is unchanged at about 3.



(a) Mean wall-clock time per workload (2 nodes, 128 GB, 32 CPUs per node, log scale) for Dask 3.11, Dask 3.13 and Spark.

(b) The same data as a Dask 3.11/3.13 ratio; the dashed line marks parity.

Figure 14: Python 3.11 versus 3.13 effect on Dask wall-clock time across workloads. Gains concentrate on Python-bound workloads (data transformation, feature engineering, clustering); BLAS-bound matrix operations and the small model-training job are unchanged. Panel (b) is a single configuration (2N/128GB/32c); cf. Table 10 for the cross-configuration mean.

Table 10: Effect of Python version on Dask wall-clock time (mean across the two-node configurations).

Benchmark	Dask py3.11	Dask py3.13	Speedup
Data transformation	384.5 s	256.0 s	1.5×
Feature engineering	58.0 s	27.8 s	2.1×
K-means	210.5 s	100.0 s	2.1×
Matrix operations	7.3 s	7.5 s	1.0×
Model training	3.1 s	3.2 s	1.0×

5.10 Scalability

Verdict. Scalability is not a global property of either framework but depends on whether the inner computation releases the Global Interpreter Lock: Spark scales positively with core count on CPU-bound workloads, Dask scales only on BLAS-bound phases and is flat on Python-loop-bound workloads, adding nodes yields sub-linear gains for both, and per-node memory and CPU topology have little effect except on Spark's matrix operations.

This subsection aggregates the scaling evidence across the suite to answer **RQ2** directly, and its central message is that the question "does framework X scale?" is ill-posed without naming the workload. The compute-scaling axis is summarised in Figure 15, which plots mean wall-clock time against total core count: Spark slopes downward on every CPU-bound workload, and Dask 3.13 is largely flat. The reason is the mechanism established workload by workload above and synthesised in Section 5.11: Spark's per-task processes turn cores into throughput on Python-bound code, whereas Dask's threads, sharing one interpreter lock, only convert cores into throughput when the inner kernel is BLAS and releases that lock. Dask's flat lines are therefore not a failure to scale in general; they are the visible signature of the lock on exactly those workloads where the lock binds. Feature engineering is the one exception to "flat": in the four-node slice, Dask 3.13's wall-clock time rises rather than plateaus as cores increase (51 s for 4×32 cores against 19–39 s elsewhere), a genuine mild regression; the two-node mean, by contrast, stays essentially flat as cores double (about 27 s at 64 total cores against about 28 s at 128). The additional worker processes in the four-node layout add task-graph coordination overhead on a workload whose kernel is Python-bound and therefore cannot absorb the extra cores as throughput.

The orthogonal resource axes reinforce the same reading. Node scaling from two to four nodes (Figure 16) yields sub-linear speed-ups for almost every workload, falling short of the ideal $2\times$, and for several workloads, optimising the runtime (for instance, upgrading the Python version) returns more than adding hardware, a pointed observation for HPC resource planning. Per-node memory has a negligible effect once it is sufficient to avoid spilling (Figure 17), the sole exception being Spark's matrix operations, which isolates CPU and shuffle cost as the binding constraint there rather than memory. CPU topology (two nodes of 64 cores versus four nodes of 32 at the same total of 128 cores; Figure 18) leaves Dask essentially insensitive and benefits Spark only mildly, on feature engineering and clustering. The aggregate answer to **RQ2** is therefore conditional: positive compute scaling where the lock is absent, flat where it binds, and diminishing returns on nodes for both, with memory and topology second-order except for the one Spark workload already identified as serialisation-bound.

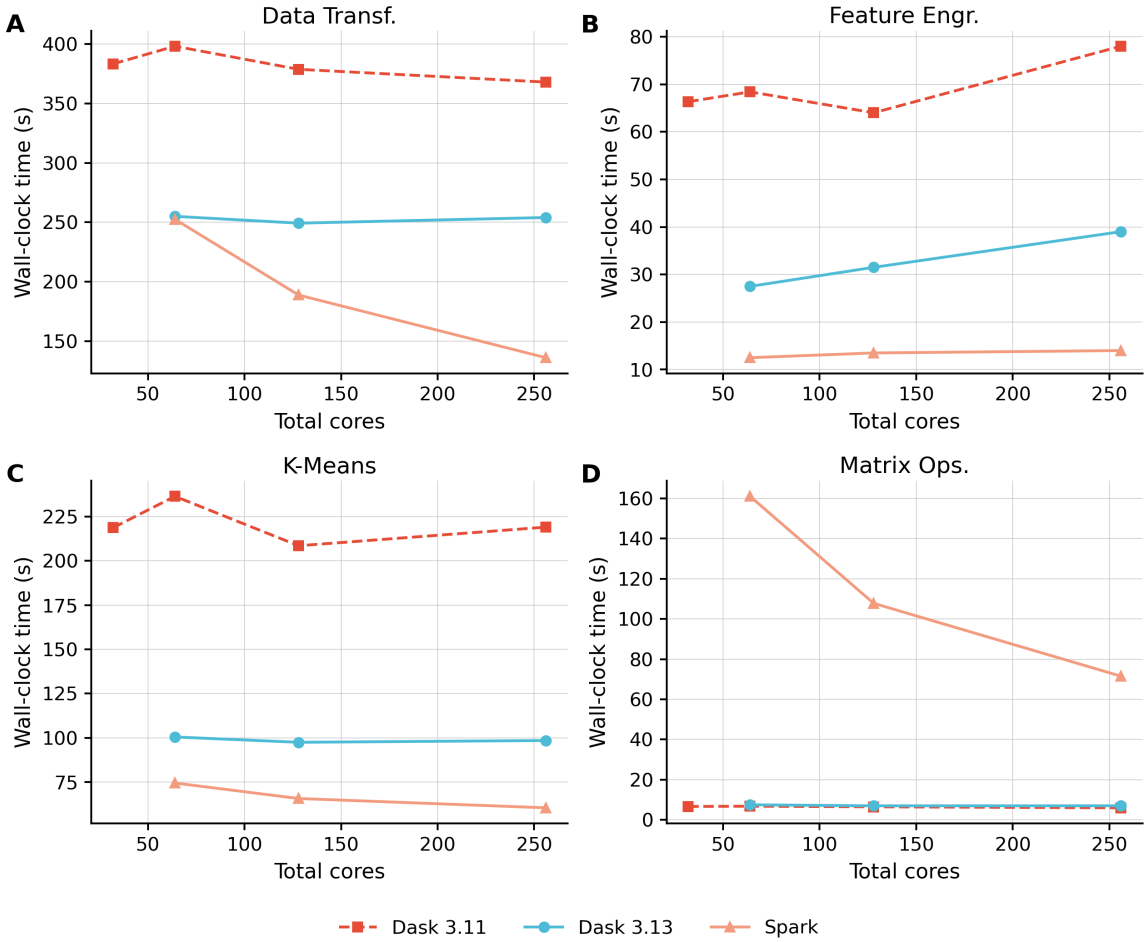


Figure 15: Scalability: mean wall-clock time versus total core count. Spark slopes down on all CPU-bound workloads; Dask 3.13 is largely flat. TF–IDF is omitted from this figure entirely: unlike the other six workloads, it was run at a single fixed configuration only (Table 8, footnote), not swept across core counts, so it has no core-count axis to plot.

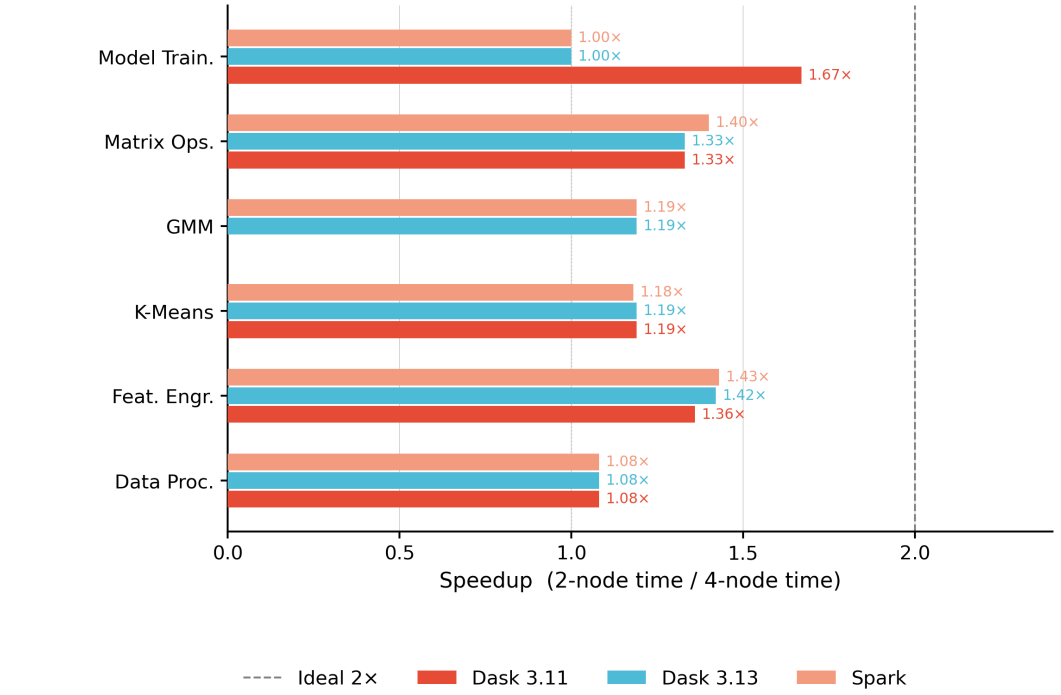


Figure 16: Node-scaling efficiency, 2→4 nodes. Almost every workload falls short of the ideal 2× (dashed). For many workloads, optimising the runtime (e.g. Python version) yields more than adding nodes.

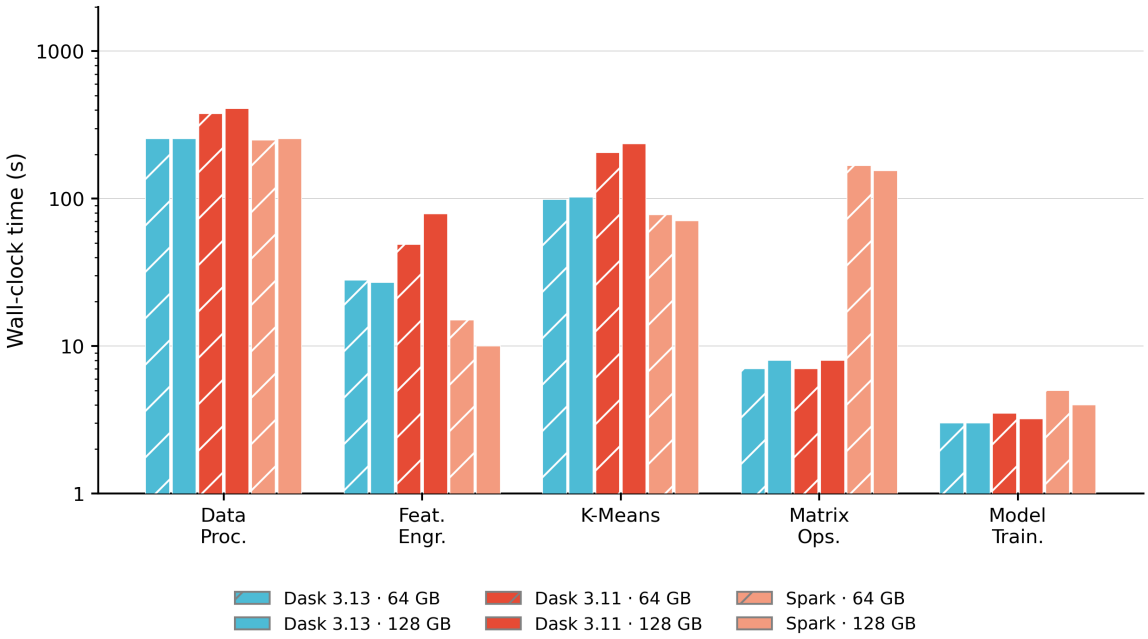
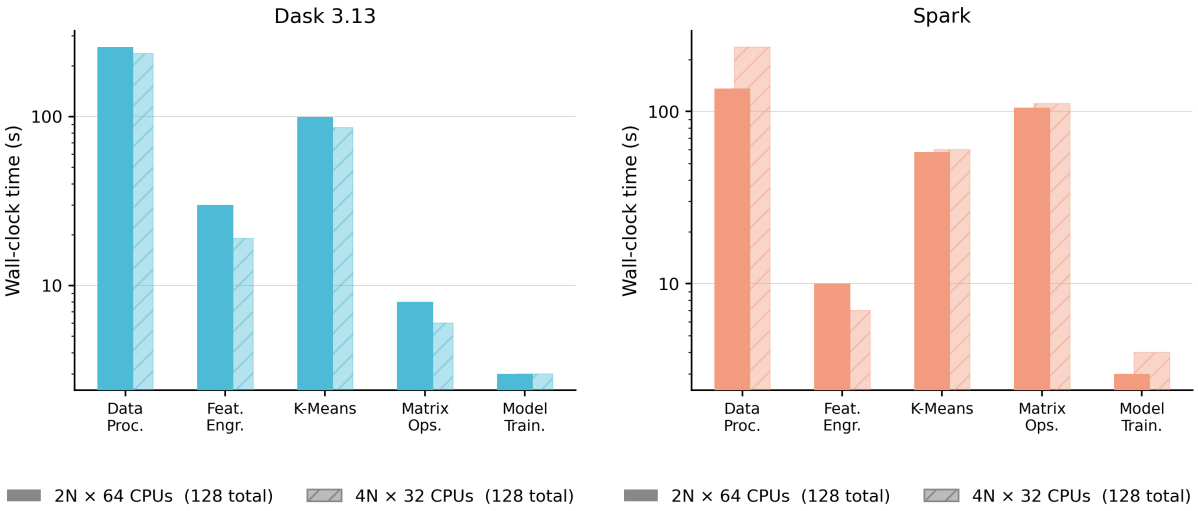


Figure 17: Memory sensitivity: 64 GB versus 128 GB per node. Per-node memory has negligible effect except on Spark matrix operations, isolating CPU/shuffle as the binding constraint.



(a) Dask 3.13 across the five tabular workloads under both layouts (log scale).

(b) Spark 3.5.1 across the same five workloads and layouts, on the same log scale.

Figure 18: CPU topology: two nodes × 64 cores versus four nodes × 32 cores at 128 total cores. Dask is modestly faster under the wider four-node layout on every workload except model training; Spark is faster only on feature engineering and regresses elsewhere, most sharply on data transformation (136 s to 236 s), where spreading the same core count over twice as many nodes converts intra-node shuffle into network traffic.

5.11 Architectural Mechanisms: Synthesis (RQ3)

The per-workload analyses converge on two cross-cutting mechanisms that, between them, account for nearly every row of Table 8. Stating them explicitly is the substance of **RQ3**: the performance differences are not a collection of unrelated quirks but the surface expression of two underlying design decisions.

Threads versus processes, and the Global Interpreter Lock. The two frameworks achieve intra-node parallelism by opposite means. In its default configuration Dask launches one worker process per node with many threads; the launch script specifies neither a worker count nor a thread count, so each of the two workers auto-sizes to the Slurm allocation, and the logs confirm two workers with 32 or 64 threads each. Those threads share one Python interpreter and therefore contend for one Global Interpreter Lock. Spark instead launches Java-virtual-machine executors of four cores each and, for a Python user-defined function, forks a separate Python worker process per task slot, so no interpreter lock is shared across cores. On any workload whose inner loop is Python-level (the data-transformation kernel, Section 5.2; the feature-engineering loop, Section 5.3; the clustering coordination, Sections 5.4–5.5), Spark's per-process model turns additional cores into genuine parallel work while Dask's threads serialise on the lock and plateau, as the effective-concurrency figure of about 6.6 quantifies. This single mechanism explains the direction of all four tabular workloads that Spark wins.

The inversion under BLAS. The interpreter lock constrains only Python-level bytecode. NumPy and BLAS kernels (matrix multiplication, singular-value decomposition, reductions) release the lock for the duration of the underlying C or Fortran computation [Har+20], so Dask's threads parallelise them fully. The very thread model that loses the data-transformation workload therefore wins matrix operations outright (Section 5.6),

and the matrix-heavy maximisation step of the Dask Gaussian-mixture routine scales with thread count for the same reason (Section 5.5). The mirror image holds for Spark: whenever a workload forces data row by row across the Python boundary, `RowMatrix` construction being the extreme case, Spark pays a per-row serialisation cost that dominates its runtime. The two mechanisms are thus a single principle viewed from two sides: *the location of the bottleneck relative to the interpreter lock decides the winner.* Where the heavy work is Python bytecode, processes beat threads and Spark wins; where the heavy work is native array code, threads beat processes and Dask wins. Every headline result in this study is an instance of that principle, which is precisely why the leaderboard generalises only through the mechanism and not on its own.

5.12 Implementation Complexity (RQ4)

Implementation complexity is treated as a first-class outcome rather than an aside, because for many practitioners the cost of *writing and maintaining* a pipeline rivals the cost of *running* it. **RQ4** is answered with the three complementary instruments defined in Section 4.7: static code metrics, an impedance-mismatch catalogue, and a structured qualitative assessment.

Static code metrics. The `radon` measurements across the twelve scripts are collected in Table 11, and they overturn the study's own initial hypothesis. The expectation was that Spark, with its more ceremonious session and typing apparatus, would require more code; the measurement shows the opposite. For the five comparable workload pairs (excluding the deliberately hand-rolled Gaussian mixture, which is discussed separately), Spark implementations are *shorter* (579 against 720 total source lines) and carry *lower* user-facing cyclomatic complexity, a mean of 3.90 against 4.68. The explanation is the same architectural fact that drives the performance results: MLlib and Catalyst absorb algorithmic and query-optimisation complexity into library calls that never appear in the user's line count. The complexity does not vanish; it *moves* out of user code and into the framework. The clearest single illustration is the Gaussian-mixture pair, where the relationship inverts: the Dask routine is far longer (230 lines against 166) precisely because the expectation–maximisation algorithm, absorbed into a single MLlib call on Spark, is written out by hand on Dask. That hand-written code is exactly what delivered the numerical-robustness advantage of Section 5.5. Higher Dask line counts elsewhere likewise reflect explicit partition-level orchestration, which buys transparency and customisability that the Spark library path does not expose.

Table 11: Static code metrics (measured via `radon 6.0.1`). SLOC = source lines of code; LLOC = logical lines of code; CC = average cyclomatic complexity per function (A = 1–5, B = 6–10); MI = maintainability index (0–100, higher is better).

Benchmark	Framework	SLOC	LLOC	CC (avg)	MI
Clustering (K-means)	Dask	144	130	5.00 (A)	43.46
	Spark	157	172	6.00 (B)	36.86
Clustering (GMM)	Dask	230	219	4.29 (A)	39.49
	Spark	166	175	6.00 (B)	36.45
Data transformation	Dask	182	150	3.67 (A)	36.83
	Spark	163	157	5.50 (B)	32.98
Feature engineering	Dask	116	108	4.75 (A)	42.30
	Spark	67	68	2.50 (A)	40.10
Matrix operations	Dask	148	145	4.75 (A)	36.91
	Spark	109	121	2.50 (A)	40.49
Model training	Dask	130	124	5.25 (B)	42.97
	Spark	83	86	3.00 (A)	46.46

The maintainability-index column tells a subtler story than the line counts. Spark's indices are not uniformly higher (on K-means, the Gaussian mixture, and data transformation the Dask scripts score better) because the maintainability index penalises the per-line Halstead volume and branching that Spark's denser, more operator-laden lines incur, even when those lines are fewer. The two metrics therefore measure different things: Spark writes fewer, heavier lines; Dask writes more, lighter ones. Neither is uniformly "simpler," which is itself the point: *simplicity* resolves into distinct, separately measurable dimensions once examined.

Impedance-mismatch catalogue. Behind the aggregate metrics lie specific, recurring structural frictions: points where one framework's data model forces code that the other does not require for the same task. Four were salient across the suite. *First, feature assembly.* Spark's MLlib estimators consume a single assembled vector column, so any multi-feature workload must route its columns through a `VectorAssembler` stage before fitting and unpack predictions afterwards; the equivalent Dask code operates on the raw array or DataFrame columns directly, with no assembly step. *Second, the user-defined-function boundary.* Applying custom per-partition logic is a `map_partitions` call receiving a pandas DataFrame on Dask, an idiom immediately familiar to any pandas user, whereas on Spark it requires either a `pandas_udf` with an explicit return-type schema or, worse for performance, a row-at-a-time Python UDF; the Spark path demands schema declarations and an awareness of the serialisation boundary that the Dask path does not. *Third, distributed matrices.* The matrix workload lays the mismatch bare: `dask.array` is a near-drop-in distributed NumPy, while Spark's `RowMatrix` and `IndexedRowMatrix` are a separate, row-oriented abstraction with their own construction ceremony and no element-wise array semantics, which is simultaneously the source of the performance gap

in Section 5.6 and of additional user code. *Fourth, resource configuration.* Spark requires manual executor sizing (the executor count and per-executor memory must be computed from the node allocation, as Section 4.5 describes), whereas Dask's workers inherit the Slurm allocation automatically; the Spark pipeline therefore carries configuration logic that has no Dask counterpart. In every case the friction traces back to the same root as the performance behaviour: Spark's JVM-and-library lineage versus Dask's native-Python-and-NumPy lineage. Proximity to the familiar pandas and NumPy idiom is the dimension on which Dask most consistently reduces implementation effort, independently of line count.

Structured qualitative assessment. The third instrument applies four fixed dimensions, namely conceptual difficulty, API familiarity, debugging surface, and migration cost, uniformly to each implementation pair, complementing the objective metrics with a controlled subjective assessment grounded in having written both sides.

The assessment is qualitative rather than tabulated: the four fixed dimensions were applied by the implementer, having written both sides of every pair, rather than scored numerically per dimension. The resulting judgement is that Dask scores better on API familiarity and migration cost, a direct consequence of its NumPy/pandas proximity documented in the catalogue above, while Spark scores better on conceptual difficulty for the workloads whose algorithm it absorbs into a single library call, notably the Gaussian mixture and the matrix operations, where the user is spared writing the algorithm at all. This qualitative finding complements the SLOC and cyclomatic-complexity results of Table 11: it is precisely the workloads where Spark's line count is lowest that its conceptual-difficulty advantage is strongest, because the same library absorption that shortens the code also removes the need to reason about the underlying algorithm.

Taken together, the three instruments answer **RQ4** without contradiction once the framing is right: Spark's library absorption yields *less user code* with *lower branching complexity*, at the cost of transparency and of the configuration and assembly ceremony catalogued above; Dask's explicitness yields *more user code* that is *closer to familiar scientific-Python idiom* and fully inspectable, at the cost of writing out machinery that Spark hides. The performance–complexity relationship this implies is the natural bridge to the discussion, made quantitative in the trade-off figure below (Figure 19): the framework writing less code is often the faster one, because the same library machinery that hides the code also optimises the execution.

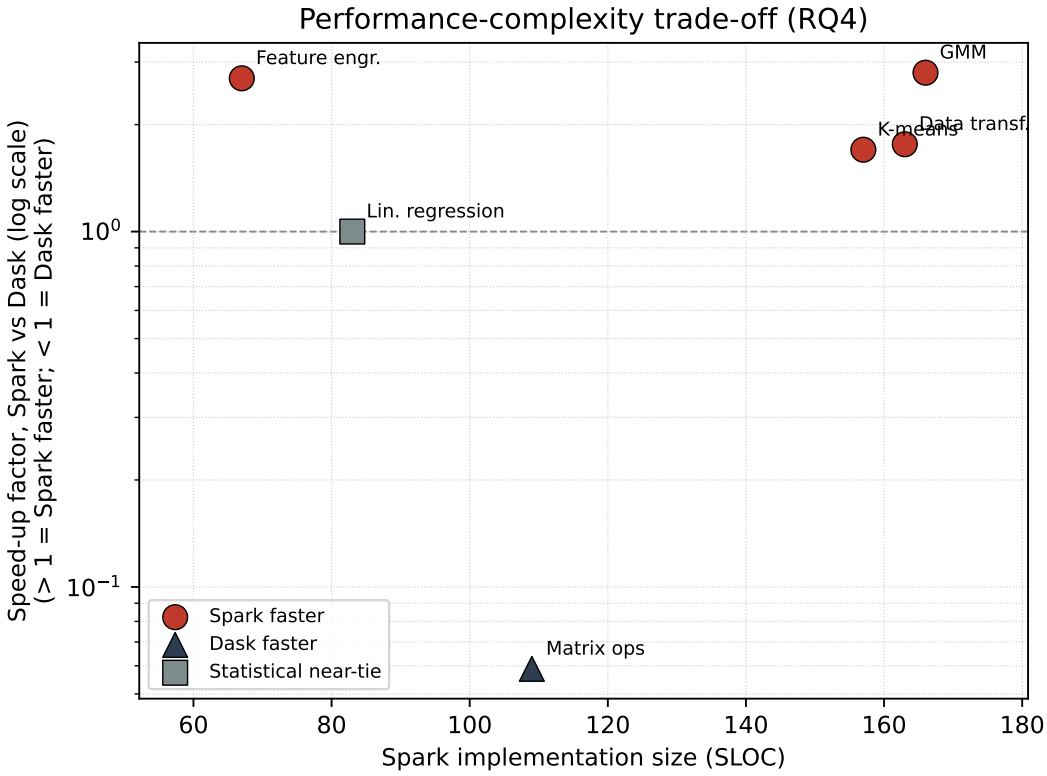


Figure 19: Performance–complexity trade-off: a scatter plot with user-facing implementation complexity (Spark SLOC from Table 11) on the x-axis and mean speed-up factor (this study's measured Spark-vs-Dask ratio per workload, log scale) on the y-axis, one point per workload, marker shape distinguishing the winning framework. Spark's wins cluster with *lower* Spark SLOC; matrix operations is the outlier, where Dask wins decisively at moderate complexity.

Table 12 closes the chapter by mapping each research question to the instrument that answered it and the evidence produced, summarising the findings detailed above before the discussion that follows interprets their broader implications.

Table 12: How each research question was answered.

RQ	Instrument (Ch. 4)	Evidence (Ch. 5)	One-line answer
RQ1	Seven-workload suite at matched allocations, repeated runs	§§5.2–5.8, Table 8, Figs. 7, 8	Split verdict: Spark 1.7–2.7× on four tabular workloads; Dask ∼17× on matrix ops, ∼10× on GPU TF–IDF; regression a tie.
RQ2	Factorial sweep over nodes, memory, cores, topology	§5.10, Figs. 15, 16–18	Conditional: Spark scales with cores where Python-bound; Dask only where BLAS-bound; node scaling sub-linear for both.
RQ3	Phase-level timing + scheduler/executor instrumentation	§5.11; per-workload §§5.2–5.8	Two mechanisms, threads-vs-processes under the GIL and its inversion under BLAS, explain nearly every result.
RQ4	Static metrics + qualitative assessment + impedance-mismatch catalogue	§5.12, Table 11	Counter-intuitively, Spark code is shorter and less complex: MLlib/Catalyst absorb complexity; Dask trades lines for transparency.

6 Discussion

This section first recapitulates the answers to the four research questions, then steps back to consider what they collectively imply, where their authority ends, the threats that qualify the conclusions, and the further work those limits motivate.

6.1 Summary of Findings

This thesis set out to answer a question that practitioners on Slurm-based HPC systems face repeatedly and that the literature had answered only partially: when should a data-science workload be run on Dask and when on Apache Spark? Rather than seek a single verdict, the work treated the comparison as workload-driven from the outset, assembling a benchmark suite that spans the data-science lifecycle along one axis and the resource-bottleneck taxonomy along another, executing it across eight hardware configurations on a production HPC cluster, and measuring not only runtime but the often-neglected dimension of implementation complexity. Four research questions (Section 1.2) structured the investigation. Each was answered by a dedicated instrument, designed for the purpose in Section 4 and applied in Section 5; Table 12, at the end of that section, states the mapping, and the paragraphs that follow summarise the substance of each answer.

On **RQ1**, relative performance, the verdict is a genuine split rather than a points-decision: Spark wins four of the six tabular workloads by factors of 1.7 to 2.7 (data transformation, §5.2; feature engineering, §5.3; K-means, §5.4; and the Gaussian mixture, §5.5), while Dask wins dense matrix operations by roughly 17× (§5.6) and GPU-accelerated text processing by roughly tenfold (§5.8), and the two reach statistical parity on linear regression (§5.7). The L1→L2 escalation embedded in **RQ1** was also answered directly: raising algorithmic intensity from K-means to the Gaussian mixture *widened* Spark's advantage

(from ∼1.7× to ∼2.3×), because the heavier per-iteration statistics reward MLlib's single tree-aggregate pass, while simultaneously surfacing Dask's compensating asset, the numerical control of the hand-written EM routine. That distribution of outcomes, stable across every configuration tested (Figure 8), is itself the headline result: no framework dominates unconditionally.

On **RQ2**, scalability, the answer is conditional and follows from the mechanism rather than from either framework's brand: Spark scales positively with core count on CPU-bound workloads (254→136 s on data transformation as cores double), Dask scales only on the phases whose kernels release the GIL into BLAS (the EM maximisation step falling 1030→491 s), both frameworks return sub-linear gains from additional nodes (Figure 16), with the pointed corollary that a software-stack upgrade can out-return a hardware increase, and per-node memory and CPU topology are second-order except for Spark's serialisation-bound matrix workload (Figures 17 and 18). Scaling, in short, is a property of the workload-on-framework pairing, not of the framework alone. This answer covers the resource-scaling axes; the complementary data-volume (weak-scaling) axis lies outside the scope of the measurements reported here, and a full experimental design for it is pre-registered in §6.5.

On **RQ3**, the architectural causes, the many per-workload findings reduce to a single principle: *the location of the bottleneck relative to the Global Interpreter Lock decides the winner*. Where the heavy work is Python bytecode, Spark's per-task processes beat Dask's lock-bound threads, quantified by Dask's effective concurrency of ≈6.6 against 64 available threads (§5.2). Where the heavy work is native array code, Dask's threads parallelise the lock-releasing kernels while Spark pays a per-row serialisation cost at the JVM-to-Python boundary, quantified by every individual Spark matrix phase exceeding Dask's entire 6-second total (Figure 11). The same thread model is therefore Dask's greatest liability and its greatest asset, and that one design decision explains nearly every row of Table 8. This is the answer's transferable form: a practitioner classifies a prospective workload against the lock, and the classification predicts the winner without re-running the suite.

On **RQ4**, implementation complexity, the measurement overturned the study's own initial hypothesis: Spark code is generally shorter (579 vs 720 SLOC over the comparable pairs) and lower in user-facing cyclomatic complexity (mean 3.90 vs 4.68), because MLlib and Catalyst absorb algorithmic and query-optimisation complexity into library calls; the complexity does not vanish but moves out of user code and into the framework. The Gaussian-mixture pair inverts the relationship (230 vs 166 lines) precisely where the algorithm is hand-written, and that hand-written code is what purchased the numerical-robustness advantage of §5.5. The maintainability index tells the complementary story: Spark writes fewer, heavier lines; Dask more, lighter ones. So "simplicity" resolves into separately measurable dimensions rather than a single ranking. The impedance-mismatch catalogue (§5.12) locates the residual friction in four recurring places: vector assembly, the UDF boundary, distributed matrices, and executor sizing.

6.2 Implications

The central implication is that the choice between Dask and Spark on Slurm-based HPC is *workload-determined rather than absolute*, and, more usefully, that the determining factor is identifiable in advance. Section 5.11 reduced the entire results table to a single principle: the winner is set by where the bottleneck sits relative to the Global Interpreter Lock. That principle is directly actionable. A practitioner who can characterise a prospective

workload as Python-bytecode-bound (a custom per-row or per-partition Python loop, a shuffle-heavy aggregation expressed through Python) should expect Spark's process-based executors and Catalyst-compiled plans to turn hardware into parallel work, and should prefer Spark. A practitioner whose workload is dense linear algebra that descends into BLAS, or who has a GPU path available for an embarrassingly parallel text or array computation, should expect Dask's thread model to parallelise the native kernels and should prefer Dask. The decision does not require running the full benchmark suite; it requires classifying the workload against the one axis this study has shown to be decisive.

The implementation-complexity findings sharpen the implication rather than complicating it. Because MLlib and Catalyst absorb algorithmic and query-optimisation complexity into library calls, Spark code is generally shorter and lower in user-facing branching complexity, but that absorption is exactly what removes the user's ability to see and modify the machinery. The practitioner therefore trades *observable, modifiable orchestration* (Dask) against *absorbed, optimised library machinery* (Spark), and the right trade depends on whether the workload's bottleneck is one the library already optimises. When it is (the Gaussian mixture, K-means, a deep feature chain), Spark offers both less code and more speed, an unusually clean win. When it is not (dense distributed matrices, or any computation requiring numerical control the library does not expose), Dask's explicitness becomes an asset on both axes at once. The two research dimensions, performance and complexity, thus point the same way for any given workload, which is what makes the workload-classification heuristic above trustworthy rather than a coin toss between competing concerns.

A third implication is addressed to HPC operations rather than to application authors. The Python-version result (Section 5.9) and the node-scaling result (Section 5.10) together show that, for several workloads, upgrading the software environment returns more than allocating additional nodes. In a shared HPC setting where node-hours are the scarce, contended resource, a 1.5-to-2.1× wall-clock improvement obtained by moving from Python 3.11 to 3.13, at zero additional hardware cost, is a more efficient lever than a node-count increase that delivers sub-linear returns. The software stack is not merely a correctness concern to be pinned and forgotten; it is a performance lever of the same order as hardware, and sometimes a larger one.

The findings of Section 5 compress into a three-question decision procedure: (1) where does the workload's bottleneck sit relative to the Python interpreter: in native/BLAS code, in Python bytecode, or in data movement; (2) does the pipeline iterate with cross-node shuffles; (3) is a GPU available and is the operation vectorisable onto it. Figure 20 renders the procedure; Table 13 summarises it with the supporting evidence.

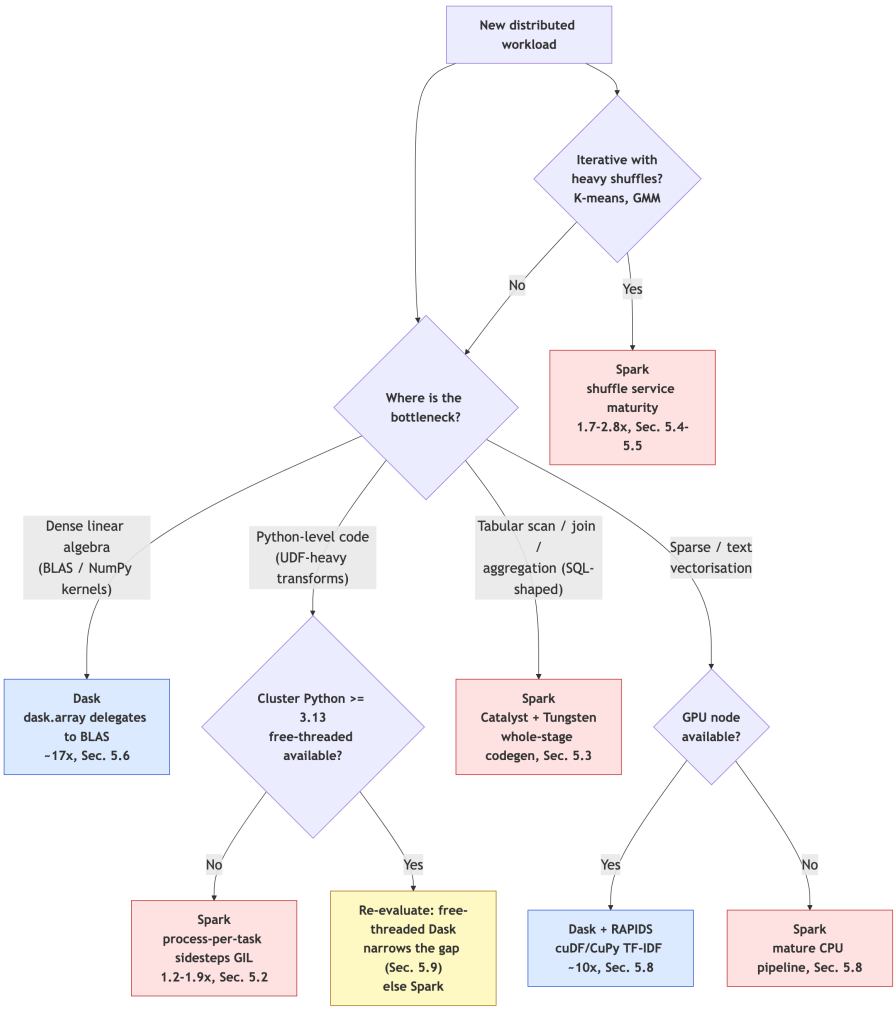


Figure 20: A decision procedure for framework selection, derived from the mechanisms of Section 5. The "free-threaded available?" branch was not evaluated here.

Table 13: Framework recommendation by workload class, derived from the mechanisms of Section 5.

Workload class	Dominant cost	Recommendation	Evidence	Caveat
ETL with Python UDFs	Python bytecode under GIL	Spark	§5.2, 1.2–1.9×	Gap narrows under free-threaded CPython (§5.9)
Column-expression feature eng.	Query planning + codegen	Spark	§5.3, ∼2.7×	Dask acceptable if pipeline stays in pandas idioms
Iterative clustering (K-means/GMM)	Shuffle + per-iteration scheduling	Spark	§5.4–5.5, 1.7–2.3×	Dask penalty grows with iteration count
Dense linear algebra	BLAS kernels	Dask	§5.6, ∼17×	Ensure chunk sizes match BLAS blocking
Small model training	Fixed startup overhead	Tie	§5.7	Framework choice immaterial below ∼10 s jobs
Text vectorisation w/ GPU	Vectorisable transform	Dask + RAPIDS	§5.8, ∼10×	Requires A100-class node; CPU-only → Spark
Team expertise dominant	Migration + maintenance cost	Incumbent framework	§5.11–5.12	Complexity gap (Ch. 5) rarely justifies a rewrite alone

Worked example 1: Shuffle-heavy ETL. A nightly job joins a 500 GB Parquet click-stream against a dimension table, aggregates by user, and writes back to the parallel filesystem. The bottleneck is the join shuffle, not Python code: the decision tree exits at the SQL-shaped branch. Recommendation: Spark; expect Catalyst to push predicates into the Parquet scan and whole-stage codegen to fuse the projection chain (§5.3 mechanism). Dask would execute the same plan without cost-based join reordering and with per-task graph overhead growing with partition count.

Worked example 2: Simulation post-processing. A research group post-processes 2 TB of gridded NumPy output with blocked SVDs and covariance computations. The arithmetic lives in BLAS; the decision tree exits at the dense-linear-algebra branch. Recommendation: Dask; `dask.array` expresses the blocked algorithm natively and §5.6 measured ∼17× at exactly this regime. Spark would pay the JVM↔Python serialisation boundary on every block (§5.6's row-oriented penalty).

Worked example 3: GPU-eligible NLP prep. A team vectorises 700 K documents with TF–IDF on a cluster that has A100 nodes idle overnight. The transform is hash-based and vectorisable; the decision tree exits at the GPU branch. Recommendation: Dask with cuDF/CuPy (§5.8, ∼10×), *provided* the GPU allocation is actually schedulable; if the queue puts the job on CPU nodes, the recommendation flips to Spark, which is the robustness dimension §5.8 reports.

6.3 Limitations

The **RQ1** verdicts are stated throughout as configuration-specific ("Spark was X× faster in this configuration") rather than as absolute, portable properties of the frameworks, because the magnitudes are measured under a single factorial resource sweep on one dataset family and one HPC cluster. The winner heatmap (Figure 8) is stable across the measured configurations, so the direction of each verdict is robust within the sweep even where the magnitude is not. This qualification does not touch the Python 3.11-versus-3.13 comparison, which ran entirely on `scc-cpu` with the hardware held fixed and is therefore the study's one fully controlled axis; this is part of why that comparison is reported separately and weighted accordingly.

The study's scope is bounded in two further respects. It uses a single HPC cluster and, for the tabular workloads, a single primary dataset family (the New York taxi records), so generalisation to other hardware or to qualitatively different data profiles is not claimed. It includes two deliberate implementation divergences, the GPU TF–IDF path and the library-versus-custom Gaussian mixture, which are reported as ecosystem-level or robustness differences rather than like-for-like speedups. Each of these is carried forward into the threats-to-validity enumeration in Section 6.4 and, where appropriate, into the future-work discussion of Section 7.

6.4 Threats to Validity

The following threats qualify the conclusions without overturning the headline outcomes. They are enumerated so that a reader can calibrate the weight each result will bear.

Deliberate implementation divergences. The GPU TF–IDF path (cuML versus CPU MLlib) and the library-versus-custom Gaussian mixture compare different implementations by design. They are reported as ecosystem-level advantages or robustness differences, never as like-for-like framework speedups, and the relevant sections flag the divergence at the point of use.

Default configurations. All results reflect out-of-the-box framework settings. The Dask thread-versus-process lever in particular would be expected to change the Python-bound outcomes; the comparison should be read as one of defaults, not of each framework's tuned ceiling.

Single cluster and dataset. Results derive from one HPC system and primarily one dataset family. Generalisation to other hardware or to data with markedly different cardinality, skew, or column structure is not claimed.

Measurement protocol. Repeated runs for model training, with cold and steady-state timings reported separately following Georges et al. [GBE07], mitigate scheduler and allocation-placement variance but do not eliminate it. The reported margins should be read with that residual variance in mind, which is one further reason the sub-second linear-regression difference is treated as a tie rather than a result.

6.5 Future Work

Two extensions would strengthen the generality of the conclusions. The first is to complete the specified data-volume (weak-scaling) experiment, varying input size at fixed resources across three data volumes drawn from the same dataset family (35, 71, and 142 million

rows; the design is given in full below), so that the **RQ2** scaling story covers the weak-scaling axis as fully as it now covers compute, node, memory, and topology scaling. The second is to broaden the empirical base beyond a single cluster and a single dataset family, adding data with different cardinality, skew, and column structure to test whether the bottleneck-classification heuristic that organises these results continues to predict the winner. Each extension sharpens a conclusion the present work states conditionally; none, on the evidence gathered here, appears likely to overturn the central principle that on Slurm-based HPC the choice between Dask and Spark is decided by where a workload's bottleneck sits relative to the Global Interpreter Lock.

A third, narrower extension follows from the TF–IDF result of Section 5.8: applying the RAPIDS Accelerator for Apache Spark (`spark-rapids`) [NVI20] to bring GPU acceleration inside Spark's own SQL engine would test whether the $\sim$10$\times$ Dask-plus-cuDF advantage measured here survives when Spark, too, is allowed a GPU. A fourth, minor item follows from the K-means result of Section 5.4: `scripts/run_dispersion.sh`, a repeated-run re-measurement at one fixed configuration reporting variance in cluster assignments for both frameworks, is specified but was not executed as part of this study, and would confirm or refute the unquantified impression that Dask's assignments vary less than Spark's.

A pre-registered weak-scaling experiment. All results reported in Section 5 hold the dataset fixed at 142 million rows and vary resources, a *strong-scaling* design in Gustafson's terminology [Gus88]. The complementary axis is a weak-scaling sweep that varies data volume at a fixed, mid-sweep resource point (2 nodes, 64 cores, 128 GB; the configuration at which the strong-scaling curves of Sections 5.2–5.5 have plateaued for both frameworks). The design is specified here in full so that it can be executed as a direct continuation of this thesis. Three dataset sizes are drawn from the same NYC-taxi Parquet family: 35 million rows ($\frac{1}{4}\times$), 71 million rows ($\frac{1}{2}\times$), and the full 142 million rows ($1\times$), preserving the 64 MB row-group layout so that partition counts scale with volume. Two workloads represent the two regimes established in Section 5: data transformation (Python-bound, GIL-sensitive) and K-means (shuffle-bound, iterative). Each cell would be run several times, with medians and min–max whiskers reported, and each workload/framework series fitted to $t \propto n^{\beta}$, where β is the exponent of a least-squares fit in log–log space ($\beta = 1$ is ideal linear scaling in data volume; $\beta > 1$ indicates super-linear cost growth from shuffle or scheduling overhead). Three falsifiable hypotheses follow from the mechanisms established in Section 5 and are stated here *before* any measurement is taken: (1) for data transformation, both frameworks should scale near-linearly ($\beta \approx 1$) because the kernel is embarrassingly parallel per partition, and Dask's flat GIL-bound plateau predicts that its *absolute* gap to Spark should stay roughly constant in relative terms across volumes; (2) for K-means, Spark's β should stay near 1 because its mature shuffle implementation amortises with scale, while Dask's β should exceed 1 as per-iteration graph-materialisation and centroid-exchange overheads grow with partition count; (3) as a crossover check, if Dask's small-data disadvantage shrinks at 35 million rows (fixed per-task overhead amortised over less work favouring the lighter scheduler), this would qualify the practitioner guidance of Section 6.2 with an explicit data-volume threshold. The value of the mechanistic explanations of Section 5 is precisely that they make such predictions before measurement; confirming or refuting them is the most direct continuation of this work.

7 Conclusion

Section 6 above recapitulated the answers to the four research questions and drew out their implications, limitations, threats to validity, and the future work they motivate. This closing section revisits the thesis's contributions and scope, then distils three take-home messages meant to outlast the specific numbers.

Contributions, revisited. The five contributions promised in Section 1.3 were delivered as follows: the Slurm-native benchmark methodology and its automation (Section 4) produced the measurements; the multi-workload empirical comparison answered **RQ1**–**RQ2** (§§5.1–5.10); the mechanistic per-workload analysis answered **RQ3** (§5.11); the implementation-complexity framework answered **RQ4** (§5.12) and is reusable beyond this thesis; and the practitioner guidance converts the results into decision support.

Scope of validity. These answers hold for the measured regime: default framework configurations, one production cluster, one primary tabular dataset family, and Spark 3.5.1 against Dask on CPython 3.11/3.13 (§6.3). The **RQ1** magnitudes are therefore configuration-specific because they are measured under one factorial resource sweep, one dataset family, and one HPC cluster; the **RQ3** mechanism, by contrast, is grounded in framework design facts that do not depend on this cluster, which is why the thesis puts its confidence in the mechanism rather than the leaderboard.

7.1 Three Take-Home Messages

First, no framework dominates unconditionally; the winner is set by which resource saturates first. Spark wins the four tabular workloads (1.7–2.7×) where CPU loops or shuffles meet the interpreter lock; Dask wins dense linear algebra (∼17×) and GPU text processing (∼10×) where BLAS or single-instruction-multiple-thread parallelism removes the lock constraint. The right question is never "which framework is faster?" but "where does this workload's bottleneck sit?"

Second, Dask's thread model is simultaneously its greatest liability and its greatest asset. One multi-threaded worker serialises Python loops, handing Spark four workloads, yet fully parallelises BLAS, handing Dask the matrix workload outright. This single design decision, more than any other, explains the shape of the results, and it is why the heuristic of classifying a workload against the interpreter lock is reliable.

Third, the software environment is a first-class experimental variable. Upgrading Dask from Python 3.11 to 3.13 improved wall-clock time by 1.5 to 2.1× on data transformation, feature engineering, and clustering, at zero hardware cost, and for several workloads exceeding the return on adding nodes.

7.2 Data and Code Availability

The benchmark implementations, extraction and plotting scripts, extracted result tables, and the LaTeX source of this thesis are maintained in a Git repository at `https://gitlab.gwdg.de/yasir.yasir/thesis-latex`. The primary dataset, the NYC Yellow Taxi Trip Records, is publicly available from the New York City Taxi and Limousine Commission [New26]; the secondary text dataset is the publicly available Amazon Reviews *CDs & Vinyl* subset [NLM19].

References

[Aln+21] Nawras Alnaasan et al. *OMB-Py: Python micro-benchmarks for evaluating performance of MPI libraries on HPC systems*. 2021. arXiv: 2110.10659.

[Apa16] Apache Arrow Development Team. *Apache Arrow: A cross-language development platform for in-memory data*. Apache Software Foundation. 2016. URL: https://arrow.apache.org (visited on 08/13/2026).

[Arm+15] Michael Armbrust et al. "Spark SQL: Relational data processing in Spark". In: *2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 1383–1394.

[AV07] David Arthur and Sergei Vassilvitskii. "k-means++: The advantages of careful seeding". In: *Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2007, pp. 1027–1035.

[Bah+12] Bahman Bahmani et al. "Scalable k-means++". In: *Proceedings of the VLDB Endowment* 5.7 (2012), pp. 622–633.

[BB20] Stanislav Böhm and Jakub Beránek. "Runtime vs scheduler: analyzing Dask's overheads". In: *IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS)*. 2020, pp. 1–8.

[Dat21] Databricks. *Benchmark: Koalas (PySpark) and Dask*. Databricks Engineering Blog. 2021. URL: https://www.databricks.com/blog/2021/04/07/benchmark-koalas-pyspark-and-dask.html (visited on 08/13/2026).

[DG04] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters". In: *6th Symposium on Operating Systems Design and Implementation (OSDI)*. 2004, pp. 137–150.

[Dug+19] Mathieu Dugré et al. "A performance comparison of Dask and Apache Spark for neuroimaging". In: *IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS)*. 2019, pp. 40–49. arXiv: 1907.13030.

[Dug+23] Mathieu Dugré et al. "Performance comparison of Dask and Apache Spark on HPC systems for neuroimaging". In: *Concurrency and Computation: Practice and Experience* 35.21 (2023), e7635. arXiv: 2406.01409.

[Dün+17] Celestine Dünner et al. "Understanding and optimizing the performance of distributed machine learning applications on Apache Spark". In: *IEEE International Conference on Big Data*. 2017, pp. 331–338. arXiv: 1612.01437.

[Err+15] Ugo Erra et al. "Approximate TF–IDF based on topic extraction from massive message stream using the GPU". In: *Information Sciences* 292 (2015), pp. 143–161.

[Esm+16] Alireza Esmaeilpour et al. "Distributed Gaussian mixture model summarization using the MapReduce framework". In: *Canadian Conference on Artificial Intelligence*. LNCS. Springer, 2016, pp. 323–335.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically rigorous Java performance evaluation". In: *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 2007, pp. 57–76.

[Gro23] Sam Gross. *PEP 703 — Making the GIL Optional in CPython*. Python Enhancement Proposals, Python Software Foundation. 2023. URL: https://peps.python.org/pep-0703/ (visited on 07/04/2026).

[Gu+15] Rong Gu et al. "Efficient large-scale distributed matrix computation with Spark (Marlin)". In: *IEEE International Conference on Big Data*. 2015, pp. 2327–2336.

[Gus88] John L. Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[GWD26] GWDG. *HPC Documentation*. Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen. 2026. URL: https://docs.hpc.gwdg.de (visited on 07/19/2026).

[Har+20] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362.

[HBK17] Sasan Harifi, Ehsan Byagowi, and Madjid Khalilian. "Comparative study of Apache Spark MLlib clustering algorithms". In: *International Conference on Data Mining and Big Data (DMBD)*. Vol. 10387. LNCS. Springer, 2017, pp. 61–73.

[HMT11] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions". In: *SIAM Review* 53.2 (2011), pp. 217–288.

[Hua+10] Shengsheng Huang et al. "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis". In: *IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*. 2010, pp. 41–51.

[LN89] Dong C. Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Mathematical Programming* 45.1–3 (1989), pp. 503–528.

[McK10] Wes McKinney. "Data structures for statistical computing in Python". In: *9th Python in Science Conference (SciPy)*. 2010, pp. 56–61.

[Men+16] Xiangrui Meng et al. "MLlib: Machine learning in Apache Spark". In: *Journal of Machine Learning Research* 17.34 (2016), pp. 1–7.

[Mon26] José Daniel Montoya Salazar. *Unlocking Python's Cores: Hardware Usage and Energy Implications of Removing the GIL*. 2026. arXiv: 2603.04782.

[Mor+18] Philipp Moritz et al. "Ray: A distributed framework for emerging AI applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 561–577.

[Myt+09] Todd Mytkowicz et al. "Producing wrong data without doing anything obviously wrong!" In: *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2009, pp. 265–276.

[Neu11] Thomas Neumann. "Efficiently compiling efficient query plans for modern hardware". In: *Proceedings of the VLDB Endowment* 4.9 (2011), pp. 539–550.

[New26] New York City Taxi and Limousine Commission. *TLC Trip Record Data*. 2026. URL: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page (visited on 07/20/2026).

[NLM19] Jianmo Ni, Jiacheng Li, and Julian McAuley. "Justifying recommendations using distantly-labeled reviews and fine-grained aspects". In: *2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2019, pp. 188–197.

[NVI20] NVIDIA. *RAPIDS Accelerator for Apache Spark*. 2020. URL: `https://nvidia.github.io/spark-rapids/` (visited on 07/04/2026).

[Oku+17] Ryosuke Okuta et al. "CuPy: A NumPy-compatible library for NVIDIA GPU calculations". In: *Workshop on Machine Learning Systems (LearningSys) at NIPS*. 2017.

[Pad+23] Vincenzo Padulano et al. "Leveraging state-of-the-art engines for large-scale data analysis in high energy physics". In: *Journal of Grid Computing* 21.4 (2023), p. 9.

[Put+17] Fadhil K. Putri et al. "DISPAQ: Distributed profitable-area query from big taxi trip data". In: *Sensors* 17.10 (2017), p. 2201.

[RAP18] RAPIDS Development Team. *RAPIDS: GPU-accelerated data science libraries*. NVIDIA. 2018. URL: `https://rapids.ai` (visited on 07/04/2026).

[RM18] Mark Raasveldt and Hannes Mühleisen. "Fair benchmarking considered difficult". In: *International Workshop on Testing Database Systems (DBTest)*. 2018, 2:1–2:6.

[RM19] Mark Raasveldt and Hannes Mühleisen. "DuckDB: An embeddable analytical database". In: *2019 International Conference on Management of Data (SIGMOD)*. 2019, pp. 1981–1984.

[Roc15] Matthew Rocklin. "Dask: Parallel computation with blocked algorithms and task scheduling". In: *14th Python in Science Conference (SciPy)*. 2015, pp. 130–136.

[Roc23] Matthew Rocklin. *Spark, Dask, DuckDB, and Polars: Benchmarks*. PyData NYC. 2023. URL: `https://www.youtube.com/watch?v=DwtbPhbDecQ` (visited on 08/13/2026).

[RPN20] Sebastian Raschka, Joshua Patterson, and Corey Nolet. "Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence". In: *Information* 11.4 (2020), p. 193.

[Scu+15] D. Sculley et al. "Hidden technical debt in machine learning systems". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2015, pp. 2503–2511.

[Sha21] Mark Shannon. *PEP 659 — Specializing Adaptive Interpreter*. Python Enhancement Proposals, Python Software Foundation. 2021. URL: `https://peps.python.org/pep-0659/` (visited on 07/04/2026).

[TK18] Anthony Thomas and Arun Kumar. "A comparative evaluation of systems for scalable linear algebra-based analytics". In: *Proceedings of the VLDB Endowment* 11.13 (2018), pp. 2168–2182.

[Wan+14] Lei Wang et al. "BigDataBench: A big data benchmark suite from internet services". In: *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014, pp. 488–499.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[Xia+20] Yingyuan Xiao et al. "A survey of parallel clustering algorithms based on Spark". In: *Scientific Programming* 2020 (2020), p. 8884926.

[YJG03] Andy B. Yoo, Morris A. Jette, and Mark Grondona. "SLURM: Simple Linux utility for resource management". In: *Job Scheduling Strategies for Parallel Processing (JSSPP)*. Vol. 2862. LNCS. Springer, 2003, pp. 44–60.

[Zad+16] Reza Bosagh Zadeh et al. "Matrix computations and optimization in Apache Spark". In: *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 31–38.

[Zah+12] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2012, pp. 15–28.

[Zah+16] Matei Zaharia et al. "Apache Spark: A unified engine for big data processing". In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[Zev+22] Jure Zevnik et al. "Comparison of distributed computing frameworks for image classification". In: *Conference on Data Mining and Data Warehouses (SiKDD)*. 2022.

[Zio+21] Alexandros Nikolaos Ziogas et al. "Productivity, portability, performance: Data-centric Python". In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2021, p. 95. arXiv: 2107.00555.

A Full Per-Configuration Results

This appendix records the per-configuration phase-level wall-clock measurements underlying the aggregated figures and tables of Section 5. Times are in seconds. The tables below report the two-node configurations across the memory $\in$ 64, 128 GB and cores $\in$ 32, 64 grid, for which complete phase breakdowns were captured; the four-node configurations follow the same pattern and are reflected in the aggregate figures. Dask figures are the Python 3.13 build unless stated. "—" denotes a configuration that failed with an out-of-memory error (the Gaussian mixture at $\leq$ 64 GB per node, for both frameworks).

A.1 K-means clustering (phase breakdown)

Framework	Config (nodes/mem/cores)	Load	Preprocess	Assemble+Scale	K-means fit	Total
Spark	2 / 128G / 32	4.06	5.41	11.34	50.44	71.24
Spark	2 / 128G / 64	4.10	4.55	9.25	40.07	57.97
Spark	2 / 64G / 32	4.19	5.46	13.03	54.91	77.60
Spark	2 / 64G / 64	4.54	4.76	11.56	42.09	62.95
Dask	2 / 128G / 32	0.01	2.93	0.90	98.29	102.12
Dask	2 / 128G / 64	0.02	2.85	0.80	95.38	99.04
Dask	2 / 64G / 32	0.02	2.81	0.86	95.55	99.24
Dask	2 / 64G / 64	0.02	2.83	0.76	96.02	99.62

The Dask fit phase is essentially flat at ∼96–98 s regardless of cores (the k-means‖ initialisation and task-graph overhead of Section 5.4); the Spark fit phase falls with cores (55→40 s).

A.2 Gaussian mixture / expectation–maximisation (phase breakdown)

Framework	Config	Load	Preprocess	Scale	Init	Fit / EM	Total
Spark	2 / 128G / 32	4.07	4.54	13.01	—	317.76	339.37
Spark	2 / 128G / 64	3.96	4.64	8.56	—	194.60	211.77
Spark	2 / 64G / 32	—	—	—	—	—	OOM
Dask	2 / 128G / 32	0.01	2.79	1.00	0.02	1029.69	1033.50
Dask	2 / 128G / 64	0.02	2.80	0.93	0.01	490.68	494.45
Dask	2 / 64G / 32	—	—	—	—	—	OOM

Both frameworks fail at 64 GB per node. The Dask routine scales with cores (1030→491 s) because its maximisation step is BLAS-bound and releases the interpreter lock (Section 5.5); Spark is ∼2.3–3.0× faster where both complete.

A.3 Data transformation (total and Dask scheduler breakdown)

Framework	Config	Total	Read Parquet	Cumulative compute	GroupBy	Chunk ops
Spark	2 / 128G / 32	254.73	—	—	—	—
Spark	2 / 128G / 64	136.23	—	—	—	—
Spark	2 / 64G / 32	249.97	—	—	—	—
Spark	2 / 64G / 64	140.32	—	—	—	—
Dask	2 / 128G / 32	254.87	9.58	1687.91	4.45	2.20
Dask	2 / 128G / 64	257.08	9.70	1695.41	4.56	2.31
Dask	2 / 64G / 32	254.58	10.63	1677.99	4.46	2.19
Dask	2 / 64G / 64	257.10	10.41	1690.43	4.47	2.27

The Dask cumulative-compute column (∼1688 s) against the ∼255 s wall-clock total is the source of the effective-concurrency figure (≈ 6.6) discussed in Section 5.2: doubling cores from 32 to 64 does not reduce the total, whereas Spark falls from ∼255 to ∼136 s.

A.4 Feature engineering, matrix operations, and model training (totals)

Table 14: Feature engineering, matrix operations, and model training: per-configuration wall-clock totals (seconds), two-node configurations.

Workload	Framework	2/128G/32	2/128G/64	2/64G/32	2/64G/64
Feature engineering	Spark	9.59	9.59	15.27	9.77
Feature engineering	Dask	26.54	30.21	28.49	25.68
Matrix operations	Spark	154.65	105.24	166.81	95.63
Matrix operations	Dask	7.60	7.58	7.21	7.26
Model training (mean of 5)	Spark	4.22	3.10	4.64	3.18
Model training (mean of 5)	Dask	2.82	2.77	2.97	2.75

Matrix-operations phase detail (Spark, 2/128G/32): Gram 50.93 s, decomposition 43.40 s, column statistics 43.83 s, each individual phase exceeding the entire Dask total of 7.60 s, confirming that `RowMatrix` construction and per-row serialisation, not the algebra, dominate (Section 5.6). Model-training run-time vectors show the cold-start signature in both frameworks (e.g. Spark 2/128G/32: 6.12, 3.93, 3.67, 3.74, 3.63 s; Dask 2/128G/32: 4.35, 2.41, 2.30, 2.54, 2.51 s), which is why the cold run is reported separately (Section 5.7).

A.5 Resource metrics (selected)

For Spark, the executor aggregates corroborate the bottleneck analysis: the data-transformation workload shows ∼10,137 s of cumulative executor time with only 0.2 MB of shuffle, confirming a compute-bound rather than shuffle-bound profile, while the Gaussian mixture shows ∼10,000 s of executor time with negligible spill. Garbage-collection time stays a small fraction of executor time across workloads (e.g. ∼12–19 s for data transformation). For Dask, peak worker resident memory for the data-transformation workload held at ∼16 GB regardless of the per-node memory cap, consistent with the workload being compute- rather than memory-bound at this scale. Figure 21 summarises the cumulative executor time, garbage-collection time, and shuffle volume across all five tabular benchmarks.

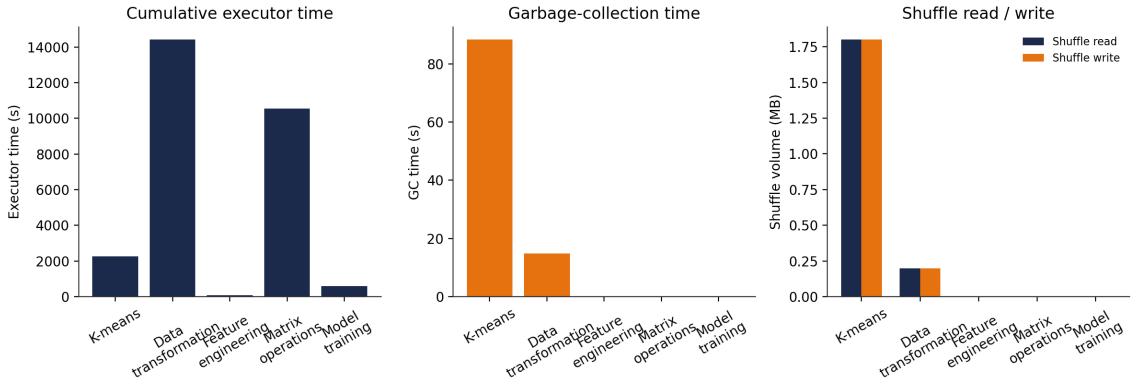


Figure 21: Spark resource-utilisation panel (2 nodes, 128 GB, 32 cores) across the five tabular benchmarks with complete stage-level instrumentation. *Left:* cumulative executor time, dominated by data transformation (≈14,400 s) and matrix operations (≈10,600 s). *Middle:* garbage-collection time, dominated by K-means (≈88 s), with a smaller amount in data transformation (≈15 s) and the remaining three workloads near zero. *Right:* shuffle read/write volume, present only for K-means (≈1.8 MB) and data transformation (≈0.2 MB); feature engineering, matrix operations, and model training shuffle negligibly at this configuration.

B Implementation and Reproducibility Notes

B.1 Framework and environment versions

The comparison used Apache Spark 3.5.1 and Dask 2026.1.1, the latter run under two CPython builds, one on 3.11 and one on 3.13, with the framework comparison reported on the 3.13 build. Static code metrics were computed with `radon` 6.0.1. The full dependency set, not merely the interpreter version, is treated as a first-class experimental variable.

B.2 Spark executor sizing

Spark executors were configured with four cores each; the executor count per node was set to (cores ÷ 4), and per-executor memory to (node memory − 8 GB) ÷ executors-per-node, reserving 8 GB per node for the operating system and daemons. At 128 GB per node this yields approximately 240 GB of aggregate executor memory across the two-node allocation, the figure cited when explaining that Spark's TF–IDF out-of-memory failures (Section 5.8) are driven by the `HashingTF` 2^{20} feature expansion rather than by a raw-memory shortage. Dask workers, by contrast, inherited the Slurm allocation automatically, which is the resource-configuration impedance mismatch catalogued in Section 5.12.

B.3 Measurement protocol

Wall-clock timing was captured per named pipeline phase. For Dask, scheduler state was snapshotted via `client.scheduler_info()` before and after each run to derive cumulative per-operation compute time and inter-worker transfer volume (the basis of the effective-concurrency figures). For Spark, the REST API (`/api/v1/applications/id/stages` and `/executors`) supplied per-stage executor time, garbage-collection time, shuffle read/write bytes, and spill metrics. Following Georges et al. [GBE07], the model-training workload was run five times per configuration with the cold first run reported separately from the steady-state mean and dispersion.

B.4 Code basis

The benchmark implementations were adapted from NVIDIA's `rapids-examples` repository, used as a code basis rather than cited as a scholarly source. All twelve benchmark scripts, the launch automation, and the analysis notebooks accompany this thesis as the released artefact (Contribution 1).

C Transparency Report: Tools and Third-Party Contributions

C.1 Overview

- Writing
  - LaTeX
- Images
  - Mermaid, rendered to PNG via `@mermaid-js/mermaid-cli`
- Diagrams
  - Python, Matplotlib
- Code

  - IDE: Visual Studio Code
  - Benchmarks run under Python 3.11 and Python 3.13
- AI
  - Claude. Used to discuss open research questions and brainstorm framing; to summarize research papers; to rephrase text and format LaTeX; to generate Python snippets for data analysis and code; and to sanity-check results. Original drafts were combined with the results data and passed to AI to improve textual coherence.

C.2 Per Section

- Section 1 Introduction
  - Used AI to review the text and improve cohesion and coherence.
- Section 4 Methodology
  - Used AI to review the text and improve cohesion and coherence.
- Section 5 Evaluation
  - Used AI to review the text and improve cohesion and coherence.
- Section 6 Discussion and Section 7 Conclusion
  - Used AI to review the text and improve cohesion and coherence.