International Journal of Computational and Experimental Science and ENgineering
(IJCESEN)

Vol. 12-No.1 (2026) pp. 1358-1365
http://www.ijcesen.com

ISSN: 2149-9144

Research Article

Reliability-First Architecture for Large-Scale Batch Data Pipelines

Akanksha Mishra*

Independent Researcher, USA
* **Corresponding Author Email:** reach.mishra.akanksha@gmail.com- **ORCID:** 0000-0002-5007-7850

Article Info:

DOI: 10.22399/ijcesen.5025
Received : 29 December 2025
Revised : 20 February 2026
Accepted : 22 February 2026

Keywords

Idempotency Patterns,
Transactional Coordination,
Temporal Consistency,
Checkpoint-Driven Recovery,
Distributed Batch Processing

Abstract:

Massive data processing systems that process data at billions of records at a time have inherent challenges in which correctness and reliability are more important than computational throughput. The reality of operations shows that instances of data quality take up a significant amount of organizational resources by taking up time in different detection stages and prolong the resolution process, which leads to cascading failures that disseminate through the downstream analytic systems. Distributed computing systems bring with them intrinsic failure modes such as the repetition of tasks via the retry mechanism, speculative execution of tasks, partial output values after multipart object storage write requests, and temporal inconsistency as part of historical reprocessing actions. The solution to these reliability issues involves architectural patterns to convert distributed uncertainty to deterministic recovery by explicit idempotency guarantees with coordination stores, transactional commit protocols where multi-object outputs are atomically published, date-scoped dependency graphs yielding temporal consistency in the process of backfills, and checkpoint-based recovery mechanisms that limit the reprocessing window. Configuration parameters that are standard to an industry, such as those provided by Apache Spark and Apache Airflow, directly determine the operational risk profile, such as settings related to the number of attempts to make on retries, speculation levels, concurrency levels, and state retention policies. The patterns of architecture shown in it set up the principles of reliability-first design, in which failures are non-destructive incidents, as opposed to disastrous. Strategies of implementation exploit coordination structures with transaction boundaries, orchestration structures with definite concurrency envelopes, and streaming structures with retention checkpoints to establish pipelines, which fail gracefully and guarantee correctness. These methods move the engineering capacity of engineering response to the engineering productive feature development by making reprocessing operations yield deterministic results consistent with the original execution semantics.

1. Introduction

As batch processing systems are scaled to process billions of records at the same time, the concern with maximizing throughput is now less important when trying to achieve correctness and reliability. The computational power to handle large-scale data is no longer the key cost factor; rather, companies are incurring growing costs to identify and rectify small-scale data quality problems before they trickle down to the final business decision and business failures. These operational challenges are quantified, with shocking precision, by recent empirical evidence of research within the industry. A survey research study performed in March 2023 with 200 data professionals found that monthly data incidents rose to 67 incidents in 2023, with 59 incidents being recorded in 2022, which is an alarming trend in data reliability issues [1]. The same study found that 68 percent of the survey participants indicated an average of 4 hours or more in incident detection times, which implies that there is a lot of delay between the time problems are detected and when teams are being informed about them [1]. Worst of all, the study recorded a 166 percent rise in time-to-resolution figures, and an average incident now takes 15 hours to completely resolve [1]. These trends were strengthened in subsequent studies that revealed two-thirds of data teams had at least one data occurrence that cost their organization more than 100,000 US dollars in the last 6 months, with 70 percent of data leaders saying that incidents took over 4 hours to be detected [2]. All these empirical results demonstrate a strong preference for reliability-first design principles: when detection takes 4 hours or longer, and resolution takes 15 hours on average per incident, pipelines designed simply to be fast, with no built-in resilience mechanisms, subject engineering teams to constant firefighting hours and decades of feature hacking instead of productive feature development and innovation [1], [2].

2. Failure Modes in Distributed Batch Systems

The nature of failure in large-scale batch processing systems has fundamental differences from simpler computational models. In contrast to the usual single-node programs that either pass or crash, distributed batch systems exist in a grey zone in which crashes are reflected by partial success, redundancy, and non-determinism. To comprehend these failure modes, it is necessary to consider two important classes: duplicate execution due to the implementation of the retry and speculation policies, and incomplete results due to the issue of distributed coordination.

The modern distributed processing models, such as Apache Spark, apply retry logic and speculative execution as a key architectural feature that will maximize throughput in the presence of transient failures and performance variability. These mechanisms, however, present subtle correctness problems where tasks do external side-effect operations. The default configuration parameters of Spark display the way duplicate execution is a normal behavioral change and not an exceptional situation. Spark registers its default value of *spark.task.maxFailures* as 4, i.e., a single task with failures may be restarted 4 times before Spark gives up on that specific unit of work, which is equivalent to 3 retries after the first run [3]. This recoverability is complicated by the speculative nature of Spark, which tries to overcome the effects of slow tasks by executing duplicate copies. The configuration parameters of interest are *spark. speculation.interval,* which is set to 100 milliseconds, *spark. speculation.Multiplier,* which is set to 3, and *spark..speculation.quantile,* which is set to 0.9 [3]. These environments guide Spark to repeatedly verify task advances after every 100 milliseconds, and upon the completion of as many tasks as possible in a phase, the framework can launch further speculative undertakings to any remaining tasks that are not running at least 3 times slower than the median completion time [3]. In computational exercises that have perfect functional purity, reading inputs, transformations, and result outputs that do not require any form of external interactions, there are no issues of correctness or problems of retry and speculation. But the instant a task has side effects like writing files to some external storage system or incrementing counters in some external database or even calling an API of some external service, the safe re-execution assumption fails in principle. Even in the absence of explicit idempotency, what would seem to be a single write operation may have the appearance of a sequence of conflicting writes to external systems, resulting in data duplication, violation of consistency, and downstream processing errors that may not be detected until much later in the pipeline execution.

The second big failure code is related to incomplete or unclear outputs to distributed storage systems, especially object storage systems such as Amazon S3, which have evolved as the new layer of persistence in large-scale data processing. The output of compute tasks can be incomplete or even have unclear semantics, even when they execute successfully and with no errors, when hundreds or thousands of partitions are writing to a common destination simultaneously. To be able to comprehend this obstacle, it is necessary to look at the technical limitations of object storage multipart upload systems. Amazon S3 can accept an object size of 48.8 tebibytes and can create as many parts per single upload operation as 10,000 multipart parts [4]. The size of each part should be between 5 megabytes and 5 gibibytes (except that the last part of an upload can be less than 5 gibibytes in size) [4]. According to Amazon Web Services documentation, developers believe that multipart uploads should be used when an object size is 100 MB and above, and that this system offers greater performance and resilience features [4]. In the case of distributed processing engines that are writing thousands of different objects representing various partitions of a dataset, failure in the middle of a commit can leave some partitions completely materialized and written to storage, and leave others missing or partially written. This results in datasets that are technically present in the storage system and can satisfy naive existence tests, but are otherwise incomplete and unsuitable for downstream products. The practice of reliability engineering should therefore incorporate a clear completeness validation logic and the atomic commit protocols, which will ensure that downstream readers do not interpret the partially written outputs as complete datasets, which are ready to be processed.

The failure detection of incomplete multipart uploads remains challenging because object storage systems provide eventual consistency for listing operations, meaning that a validation check executed immediately after a write operation may not detect missing partitions due to propagation delays that can range from milliseconds to several seconds [4]. Production monitoring systems must therefore implement delayed validation patterns with configurable wait periods, typically 30-60 seconds, before asserting dataset completeness to account for eventual consistency windows. Furthermore, orphaned multipart upload parts that remain from aborted operations consume storage costs without contributing to valid datasets, necessitating lifecycle policies that automatically clean up incomplete uploads after a defined retention period, commonly set to 7 days in high-throughput production environments.

3. Idempotency and Transactional Coordination

The realistic engineering goal of the reliable distributed system is not the absence of failures, which is an impossible task in the context of the size and complexity of the infrastructure today, but the ability to design a system in a manner capable of failing without breaking correctness and retrying without causing either non-replicated side effects or non-recyclable states. The philosophy of this design focuses on two patterns that complement each other: idempotency mechanisms that prevent individual operations from being repeated safely and transactional coordination protocols that guarantee multi-step processes that offer atomic commits.

Making distributed batch processing idempotent often requires each unit of work to be assigned a deterministic idempotency key that unambiguously identifies that work unit in all retries and a durable coordination store to protect side effects and avoid duplication of work. The work unit may be an individual division of data, a particular date of processing, or even a table as a whole, depending on the level of division that the system needs. The processing logic will refer to the coordination store before making any side effect operations, such as writing output files or making external API calls, to determine whether this idempotency key has already been processed. In case a previous execution has already done this work successfully, the side effect can safely be skipped by the current execution and continued or terminated. In case there had been no previous execution, this work, the present execution, would carry out the side effect and forbid the occurrence of an identical key by atomically storing the coordination store with the idempotency key. Amazon DynamoDB is a good option to use as a coordination store because of its high consistency guarantees and predictable performance properties. The real-world limitations of DynamoDB guide real-world design choices concerning idempotency markers: a DynamoDB item has a hard size limit of 400 kilobytes, which is enough to store an idempotency marker together with the appropriate metadata like execution status indicators, pointers to the actual data in object storage, and cryptographic checksums to verify integrity [5]. This size is generous enough to support common coordination metadata but is small enough to avoid slow read and write access, which can be a bottleneck in high-throughput pipelines.

The performance characteristics of DynamoDB conditional writes provide the necessary atomicity for idempotency guarantees through the ConditionExpression parameter, which enables check-and-set operations that prevent race conditions when multiple execution attempts attempt to claim the same work unit simultaneously, with conditional write latencies typically under 10 milliseconds at the 99th percentile for single-digit kilobyte items [6]. Cost optimization requires careful consideration of the read-write ratio in coordination stores, as DynamoDB charges are based on request units with on-demand pricing starting at 1.25 USD per million write requests and 0.25 USD per million read requests, making it economically viable to perform idempotency checks for processing tasks that execute for more than a few seconds. The choice of partition key design in the coordination store directly impacts throughput limits, with DynamoDB supporting up to 1,000 write request units per second per partition key and 3,000 read request units per second per partition key, necessitating high-cardinality key designs that distribute idempotency markers across many partitions to avoid hot partition throttling in high-concurrency scenarios.

In more complicated situations where the outputs of multiple objectives are potentially represented by a single logical dataset of hundreds or thousands of files, idempotency is not sufficient; instead, the system must provide transactional coordination so that all the outputs are visible to the downstream consumers at once, or none. This is necessary to ensure that readers cannot see half-baked datasets that might cause wrong analysis outcomes. The atomic coordination primitive is available in DynamoDB through transaction support. Grouping: The TransactWriteItems API operation provides the ability to combine the maximum number of 100 separate action requests into one atomic transaction, but the size of a single transaction of all items must not exceed 4 megabytes [6]. A strong transactional publication pattern is one that has both immutable data write and atomic metadata commitment. The first step is to write data files to a run-scoped staging prefix in object storage so that each execution attempt writes to a distinct address based on a run identifier so that different attempts do not interfere with each other. Second, after all the data files have been successfully written, the processing logic calculates a detailed manifest that includes all the anticipated partitions and files, as well as the cryptographic checksums of the data to determine integrity. Third, the manifest pointer and a status marker of "COMMITTED" are set to DynamoDB through a TransactWriteItems operation, and this ensures that such a publication occurs atomically with the 4 MB transaction size limit [6]. Lastly, downstream readers will be set to only consume data upon being referenced by a committed manifest marker and ignore any staged data files that do not have matching committed metadata. This architectural pattern is used to make retries an idempotent decision process: when a retry attempt determines that a commit marker has been left on its target work unit, it can safely terminate without trying to republish output to avoid the possibility of two or more execution attempts descending into the same downstream system with the same transient failure or speculative execution.

4. Temporal Consistency in Backfill Operations

The reprocessing of historical periods of data to add the newer logic, rectify the bugs found, or fill newly created datasets are all cases where latent errors of correctness are made visible as high-profile organizational events. A data pipeline can give the right answers to the processing today and, at the same time, can give the wrong answers when reprocessing the past, when that processing logic implicitly relies on the current state of the data instead of the historical state of the data. To design pipelines that are right in backfill, time is to be treated not as an implicit environmental variable, but as a first-class architectural consideration.

Backfill-safe pipeline architecture requires all the data dependencies to be an explicit key of the processing date or versioned snapshot identifier instead of implicitly pointing to the current state. A pipeline will be required to read the version of reference data, such as customer attributes, product catalogs, or geographic mappings, that existed on the date of the target processing date, rather than the version of the reference data that is presently available. Otherwise, a rerun of the pipeline to a date before the date of the query will silently add reference data to that date and will calculate a different result than the original processing would have given. This temporal consistency is not only on reference data but also on all inputs: fact tables should be read at partition granularity equal to the processing date, configuration parameters should be versioned and related to particular date ranges, and even system behavior (such as aggregation logic or filtering rules) must be clearly versioned to avoid unintended semantic drift throughout reprocessing.

The configuration of the orchestration system has a direct influence on operational risk in large-scale backfill operations mediated by parameters that regulate task fan-out and execution concurrency. The Apache Airflow, a common workflow orchestration platform, has multiple default configuration parameters, which define the execution envelope in the backfill cases. The *max_active_runs_per_dag* is set to the default 16, which restricts the number of active runs of the DAG that can be active at the same time within a single definition of a DAG [7]. The parameter *max_active_runs_per_dag* is also defaulted as 16 that restricts the number of task instances that are allowed to run at a time in a particular single run of a DAG [7]. The *max_map_length* default is 1024, which constrains the fan-out that may be caused by dynamic task mapping when the tasks are being generated out of XCom lists or dictionaries [7]. These integer limits have important operational consequences since the backfill processes are often many times many dates and many partitions, meaning that these task graphs are enormous. Unless the pipeline dependency graphs are actively capped and delimited with a date granularity, the backfill operations may consume the scheduler capacity and the connection pools and result in partial completion states with some dates being successful and the rest pending or failed, and they will need manual intervention to bring about consistency.

When very large sizes are involved, temporal correctness has become more of a best practice than an absolute requirement. The example of the telemetry an entertainment Platform offers a real-world illustration of the magnitude at which minor issues get magnified exponentially. According to public documentation of the logging system used at the entertainment, cardinality issues, such as a maximum of 25,000 unique tag keys an hour and tens of millions of unique tag values, can be experienced, scale-related issues that can easily reduce the query throughput and cause a huge amount of fallout when a reprocessing is needed [8]. This scale of reality is the reason why reliability-first pipeline architectures often impose severe patterns: date-partitioned inputs and outputs across the entire pipeline without exception, versioned reference data with explicit date ranges of correctness, and publication gates that do not permit downstream visibility prior to validation checks finding completeness and correctness. These patterns introduce operational complexity and decrease the maximum theoretical throughput but guarantee that backfill operations yield correct and deterministic results instead of causing cascading incidents.

5. Checkpoint-Driven Recovery Mechanisms

The nature of reliability is enhanced significantly when the ability to restart a system in a known-good intermediate state is available instead of going through the entire system every time of failure. Checkpoint mechanisms impose a limit on recovery time and reduce wasted computation by allowing recovery to restart at the last successful checkpoint instead of the initial application of a job. The difficulty is how to set the checkpoint frequency: checkpointing too often causes too much rework to do on recovery, and checkpointing too rarely causes overhead that adversely affects throughput and storage costs.

The Structured streaming subsystem of Apache Spark offers published latency envelopes, helping to design the checkpoint strategy. End-to-end processing latency can be reduced to 100 milliseconds, and the exactly-once fault-tolerance guarantees are obtained with a default micro-batch execution mode, which supports checkpoint and write-ahead logging systems [9]. Applications whose latency needs are even lower can use Continuous Processing mode, with processing latency down to 1 milliseconds (although with a weakening to at-least-once delivery guarantees, instead of exactly-once [9]. Although these figures characterize streaming workloads, the same principles are applicable to the case of batch-style incremental pipelines: checkpoint often enough so that download recovery occurs within operational recovery time limits, but not often enough to cause checkpoint overhead to dominate operational recovery time.

The extent to which a system can recover without re-computing original source data, to which checkpoint retention policies are directly related. The Spark configuration parameter spark.sql.streaming. minBatchesToRetain is set to 100 by default and determines the number of micro-batches of checkpoint history to retain by the system, which consequently determines the temporal replay window in which recovery operations can be undertaken [10]. A reliability-first pipeline does not accept this parameter as a technical default passively but as a point of service-level goals to be purposefully designed. When the operational recovery policies need the capability to re-drive a few hours of data upon the occurrence of a fault, the checkpoint retention and the checkpoint cadence should be set up in a manner that will allow the replay of a few hours of historical data within the operational time and resource limits. This involves having to multiply the checkpoint interval and retention counts in order to guarantee sufficient coverage and then test it as part of recovery exercises that periodically recover the data at an acceptable rate. Storage costs of checkpoints should also be taken into account: the higher the amount of checkpoint history stored, the more storage is used, which results in a three-way tradeoff of recovery capability, storage costs, and reprocessing overhead that should be balanced in individual operational requirements.

Table 1: *Idempotency and Transactional Coordination Patterns for Distributed Batch Systems [5, 6]*

Pattern Component	Technology	Configuration Parameter	Limit/Value	Purpose
Idempotency Marker Storage	DynamoDB	Item size limit	400 KB	Store idempotency key with metadata including status, output pointers, and checksums
Transactional Commit Fence	DynamoDB	Transact WriteItems actions	100 actions per transaction	Group multiple coordination updates into an atomic operation
Transactional Commit Fence	DynamoDB	Transaction aggregate size	4 MB	Limit total data size across all items in a single transaction
Commit Protocol Step 1	Object Storage	Data file staging	Run-scoped prefix	Write immutable outputs to an isolated staging location
Commit Protocol Step 2	Processing Logic	Manifest computation	List of partitions/files and checksums	Calculate the complete dataset inventory before publication
Commit Protocol Step 3	DynamoDB	Atomic commit marker	Single Transact WriteItems call	Publish manifest pointer and COMMITTED status atomically
Reader Safety	Downstream Systems	Consumption filter	Only committed manifests	Prevent reading from uncommitted or partial datasets

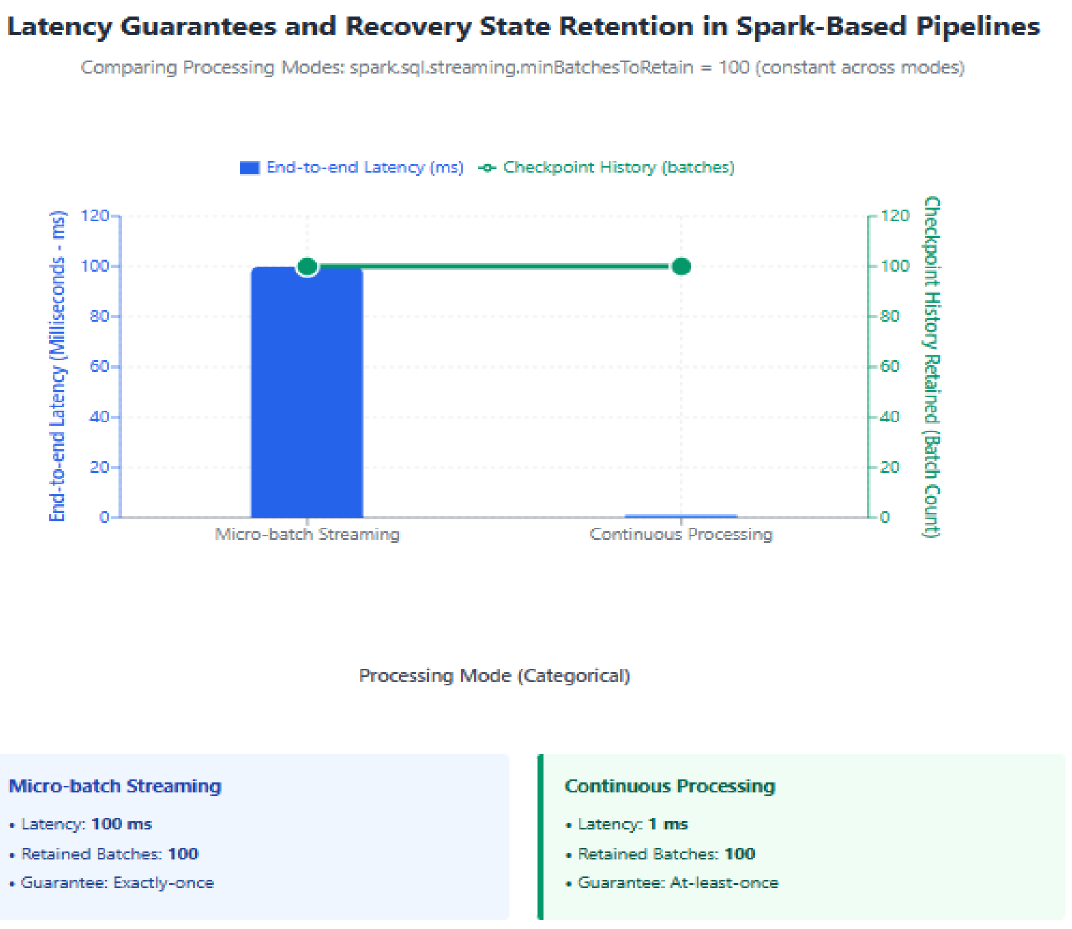

Figure 1: *Impact of Spark Retry and Speculation Parameters on Duplicate Task Execution [3, 4]*

Table 2: *Orchestration Configuration Parameters for Backfill-Safe Pipeline Architecture [7, 8]*

Backfill Safety Component	**Technology**	**Configuration Parameter**	**Default Value**	**Operational Impact**
DAG Concurrency Control	Apache Airflow	max_active_runs_per_dag	16 concurrent runs	Limits simultaneous DAG executions to prevent scheduler overload during backfills
Task Concurrency Control	Apache Airflow	max_active_tasks_per_dag	16 concurrent tasks	Bounds parallel task execution within a single DAG run
Dynamic Task Fan-out Control	Apache Airflow	max_map_length	1024 mapped tasks	Prevents unbounded expansion when generating tasks from XCom data structures
Temporal Dependency Design	Pipeline Architecture	Date parameterization	All inputs/outputs	Ensures every dependency keyed by processing date, the current state
Reference Data Versioning	Data Architecture	Snapshot versioning	Date-valid ranges	Reads reference data valid for the target processing date during backfills
Cardinality Management	Logging Systems	Unique tag keys per hour	25,000 keys	High cardinality amplifies reprocessing costs and query degradation
Cardinality Management	Logging Systems	Unique tag values	Tens of millions	Massive value spaces require careful backfill scope planning

Publication Gate	Pipeline Architecture	Validation before commit	Completeness checks	Prevents partial backfill results from reaching downstream consumers

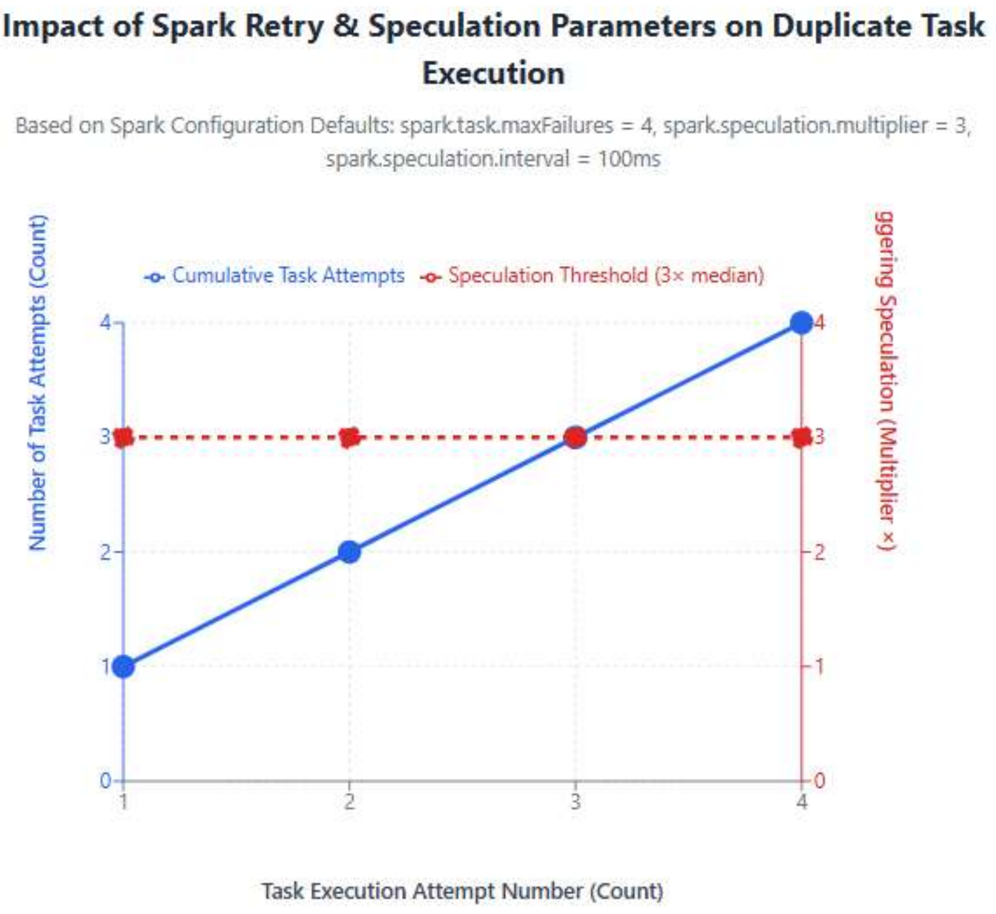

Figure 2: *Latency Guarantees and Recovery State Retention in Spark-Based Pipelines [9, 10]*

6. Conclusions

The concept of reliability engineering of large-scale batch data pipelines fundamentally changes the way organizations go about distributed data processing, since organizations heavily emphasize correctness over raw throughput. The architectural designs and operational procedures described in this material prove that today's data infrastructure should not allow failures to become an unprecedented situation that requires urgent actions but a dignified management. Explicit idempotency by use of coordination stores eliminates idempotent side effects in the event that task A fails and is retried several times by the retry logic. Atomic operation-based transactional commit protocols provide downstream consumers with either full datasets or nothing and exclude ambiguous intermediate states that distort analytical outputs. Temporal consistency Date-parametered dependency graphs with versioned reference data preserve their dependencies through backfill operations such that historical reprocessing yields the same results as when originally executed. Checkpoint-based recovery with adjustable retention intervals limits the reprocessing cost and allows the recovery to start quickly at known-good intermediate states. The parameters of configuration, such as the limits of retries and speculation, bounds of concurrency, and retention of states, have a direct impact on the operational risk profiles and recovery abilities. Organizations with these principles of reliability-first report quantifiable advantages, in addition to the lower rate of incidents: the engineering team ignores time spent in the debugging mode in favor of constructing new analytical systems and data products. The move to deterministic high-throughput systems instead of the fragile high-throughput pipelines makes it possible to scale infrastructure data without similarly increasing the operational support requirements of the infrastructure. Graceful failure of distributed batch systems with limited recovery times and assured precision provides organizations with the potential to maximize the value of their data assets by providing reliable, repeatable processing, which allows making business decisions with confidence.

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** The authors declare that they have equal right on this paper.
- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.
- **Use of AI Tools:** The author(s) declare that no generative AI or AI-assisted technologies were used in the writing process of this manuscript.

References

[1] Michael Segner, "The Annual State of Data Quality Survey," Monte Carlo, 2023. [Online]. Available: https://www.montecarlodata.com/blog-data-quality-survey.

[2] Wakefield Research, "The State of Reliable AI Survey 2024," Monte Carlo. [Online]. Available: https://info.montecarlodata.com/hubfs/Assets%20-%20Guides%2C%20Ebooks%2C%20Reports/Wakefield%20Report%20-%20State%20of%20Reliable%20AI%20Survey%202024.pdf.

[3] Apache Spark, "Spark Configuration." [Online]. Available: https://spark.apache.org/docs/latest/configuration.html.

[4] Amazon Web Services, "Amazon S3 multipart upload limits." [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/qfacts.html.

[5] Amazon Web Services, "Best practices for storing large items and attributes in DynamoDB." [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-use-s3-too.html.

[6] Amazon Web Services, "TransactWriteItems." [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_TransactWriteItems.html. Accessed: Jan. 30, 2026.

[7] Apache Software Foundation, "Configuration Reference," Apache Airflow. [Online]. Available: https://airflow.apache.org/docs/apache-airflow/stable/configurations-ref.html. Accessed: Jan. 30, 2026.

[8] ClickHouse, "How Netflix optimized its petabyte-scale logging system with ClickHouse," 2025. [Online]. Available: https://clickhouse.com/blog/netflix-petabyte-scale-logging.

[9] Apache Software Foundation, "Structured Streaming Programming Guide (Overview)," Apache Spark. [Online]. Available: https://spark.apache.org/docs/latest/streaming/index.html. Accessed: Jan. 30, 2026.

[10] Angela Chu and Tristen Wentling, "Streaming in Production: Collected Best Practices," Databricks, 2022. [Online]. Available: https://www.databricks.com/blog/streaming-production-collected-best-practices