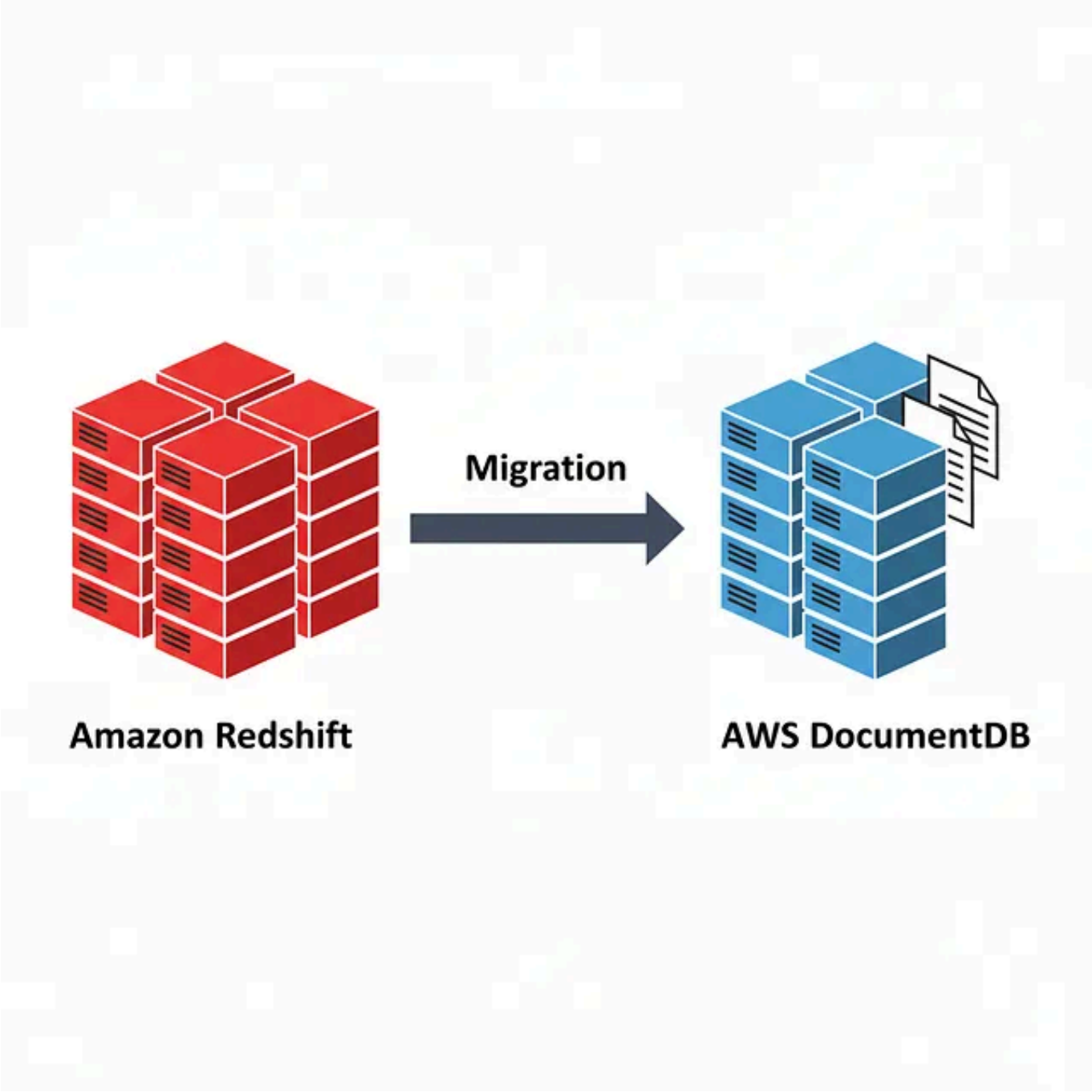


How We Shifted Our Complete Reporting DB to AWS DocumentDB from Redshift

Anurag Sharma 7 min read · Sep 15, 2025

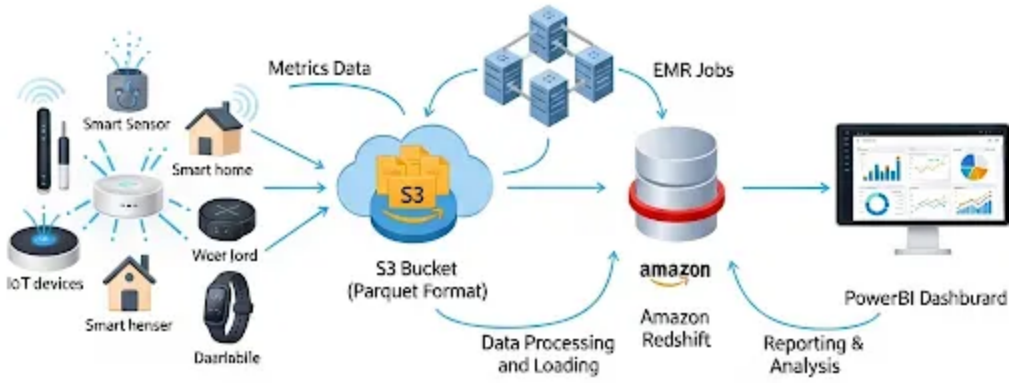
1

In the fast-paced world of an analytics company, delivering timely, accurate, and insightful reports is the lifeblood of our operations. Business users and customers rely on us for summary dashboards that provide high-level overviews, as well as the ability to drill down into granular details for deeper analysis. For years, our entire reporting ecosystem was centered around Amazon Redshift, a powerful data warehouse that served us well. However, as our data volumes exploded, we hit scalability walls that forced us to rethink our approach. This blog chronicles our journey migrating to AWS DocumentDB (MongoDB-compatible), the strategies we employed to minimize disruptions, and the innovative solutions that made the transition a success. We'll dive into the architecture, challenges, and lessons learned, offering insights for anyone considering a similar shift.

The Original Architecture: Built for Stability, But Not Infinite Scale

Our reporting pipeline was a well-oiled machine designed around Redshift's columnar storage and massively parallel processing capabilities. Data flowed in from IoT devices — think sensors collecting real-time metrics on everything from environmental conditions to equipment performance —

into Amazon Kinesis Firehose for streaming ingestion. From there, it landed in S3 buckets as raw, partitioned files (often in Parquet format for efficiency). Batch jobs, orchestrated via AWS Glue or Airflow, would then ETL (Extract, Transform, Load) this data into Redshift clusters.



This setup excelled for five to six years, handling petabyte-scale data with relative ease. Redshift's SQL compatibility allowed our analysts to write complex queries for aggregations, joins, and window functions. We optimized with sort keys, distribution styles (e.g., EVEN or KEY), and vacuum/analyze routines to maintain performance. Security was tight with IAM roles, VPC peering, and column-level access controls. Costs were managed through reserved instances and concurrency scaling, but as data grew exponentially — driven by more devices and higher-resolution telemetry — the cracks began to show. Query times ballooned from seconds to minutes, even for simple reports, and maintenance windows stretched longer despite our DBA's best efforts with deep optimizations like rewriting queries to use materialized views or partitioning tables by date.

The Tipping Point: When Redshift Started to Falter

The slowdowns weren't just annoying; they impacted SLAs with customers who expected near-real-time insights. Weekly maintenance — vacuuming

deleted rows, analyzing statistics, and resizing clusters — became insufficient as tables ballooned to billions of rows. Unused tables and reports lingered, consuming storage and compute during daily batches, inflating our AWS bills unnecessarily. Large, monolithic table structures dragged in extraneous columns from upstream sources, leading to inefficient scans.

On-the-fly processing was another culprit: Joins and ETL logic executed at query time in Redshift bogged down the system during peak hours. Our search features relied on business keys, but without proper indexing, they turned into full-table scans. As data volumes hit the terabyte mark per table, concurrency limits were hit, causing queueing and failures. We knew we needed a more flexible, document-oriented approach to handle semi-structured IoT data without the rigidity of relational schemas.

The Decision to Migrate: Embracing DocumentDB for Agility and Cost Savings

We chose AWS DocumentDB for its MongoDB compatibility, which allowed us to leverage NoSQL's schema flexibility for our varied IoT data. Unlike Redshift's row/column-oriented design, DocumentDB's document model let us store nested structures natively, reducing the need for complex joins. To keep loads light on DocumentDB, we adopted a hybrid approach: Pre-process and denormalize data in S3 using Spark jobs, storing only what's essential in DocumentDB for fast reads.

Key principles to avoid Redshift's mistakes:

- **Purge the Unused:** We audited reports and tables, dropping 30% of legacy ones that hadn't been queried in months, freeing up resources.

- **Break Down Monoliths:** Large tables were split into smaller, purpose-built collections. For example, instead of pulling all upstream columns, we selected only relevant fields, reducing data footprint by 50%.
- **Pre-Compute Everything:** No more on-the-fly ETL. Aggregations, joins, and summaries were materialized in S3 or directly in DocumentDB collections during batch runs, ensuring reports were "query-ready."
- **Hashing for Efficiency:** Retained MD5 hashing for business keys but embedded full documents instead of tilde-separated strings, allowing MongoDB's indexing to shine.

This shift promised sub-second query times, auto-scaling replicas, and better handling of unstructured data like JSON payloads from IoT.

Here's a sample of how data looked in Document DB for employee_tbl:

business_key_hash (MD5)	business_key	data_column
d41d8cd98f00b204e9800998ecf8427e	12345	John Doe~Engineering~85000~2020-05-15~Active~New York~john.doe@example.com~Senior Engineer
0cc175b9c0f1b6a831c399e269772661	67890	Jane Smith~Marketing~95000~2018-03-22~Active~San Francisco~jane.smith@example.com~Director

This format minimized joins, but parsing the data_column in queries was computationally expensive at scale.

Overcoming Migration Challenges: From Initial Hiccups to Robust Solutions

Migration wasn't seamless; we faced hurdles in data consistency, timeliness, and flexibility. Here's an expanded look at how we tackled them, including technical deep dives and trade-offs.

1. **Dual Data Availability (Redshift and DocumentDB):** Stakeholders worried about abandoning Redshift cold-turkey, so we implemented bidirectional syncing. Using AWS Database Migration Service (DMS) with change data capture (CDC), we replicated schemas in real-time for key datasets. Custom scripts handled schema mapping — Redshift's tables to DocumentDB's collections — ensuring data types aligned (e.g., converting SQL timestamps to BSON dates). This hybrid phase lasted three months, with A/B testing of reports to validate accuracy. Trade-off: Added latency for writes, but queries could route via an API gateway based on user preference.
2. **Immediate Loads or Report Refreshes:** Daily PST batches were too rigid for urgent needs, like executive requests during off-hours. We built on-demand capabilities with AWS Lambda triggers from our UI, spawning Spark jobs on EMR. These jobs pulled delta data from Kinesis or S3, applied transformations (e.g., using PySpark for filtering), and upserted via MongoDB bulk operations. We implemented idempotency with unique job IDs to avoid duplicates. Rate limiting via AWS WAF prevented overload, and monitoring with CloudWatch tracked job durations, alerting on anomalies. This reduced refresh times from 24 hours to under 10 minutes for most reports.
3. **Custom Timeframe Data Delivery:** Users often needed non-standard ranges, like "last fiscal quarter" or "year-over-year comparisons." We parameterized Spark ETL with date filters, sourcing from S3's partitioned archives (e.g., by year/month/day). Jobs used window functions in Spark SQL to aggregate dynamically, storing results in ephemeral DocumentDB

collections with TTL indexes for auto-cleanup after 24 hours. For edge cases like overlapping ranges, we added caching in Redis to reuse computations. This ensured delivery without permanent storage bloat, maintaining compliance with data retention policies.

Beyond these, we addressed data validation (using Great Expectations for schema checks), rollback mechanisms (snapshotting DocumentDB before migrations), and performance tuning (indexing on hashed keys and frequent fields).

Empowering Users: The Ad-Hoc Pipeline as a Game-Changer

Over the daily batch pipeline (running at 3 AM PST via AWS Step Functions), we layered an ad-hoc system. Users submitted requests via a custom UI built with React, Node.js backend, and AWS Cognito for auth. Submissions triggered Spark jobs on EMR, scalable to handle “n number of things” with modular plugins. We expanded its capabilities extensively:

- **Data Enrichment and Transformation:** Integrated with external APIs (rate-limited) for additions like weather data correlations, using Spark’s UDFs for custom logic. ML integrations via MLlib flagged outliers in IoT streams.
- **Backfilling Historical Data:** Handled corrections by replaying S3 logs with Spark Streaming, updating DocumentDB atomically to maintain consistency.
- **Export and Sharing Options:** Supported multiple formats with compression (e.g., GZIP’d CSV), uploaded to S3 with lifecycle policies. Integrated with Amazon SES for emailed attachments.

- **Auditing and Compliance Checks:** Logged to Amazon Elasticsearch for searchable audits, enforcing GDPR-like anonymization where needed.
- **Scalable Bulk Operations:** Auto-scaled clusters based on input size estimators, using Spot Instances for cost savings (up to 70% cheaper).
- **Integration with Downstream Services:** Webhooks pushed data to BI tools like Power BI or Slack bots for instant notifications.
- **Error Handling and Retries:** Exponential backoff retries with dead-letter queues in SQS, plus dashboards in Grafana for job forensics.
- **Advanced Analytics:** Added support for graph queries (e.g., device relationships) using Spark GraphX, storing results as embedded docs.
- **Security Enhancements:** Role-based access ensured only authorized users could trigger sensitive jobs, with encryption at rest/transit.

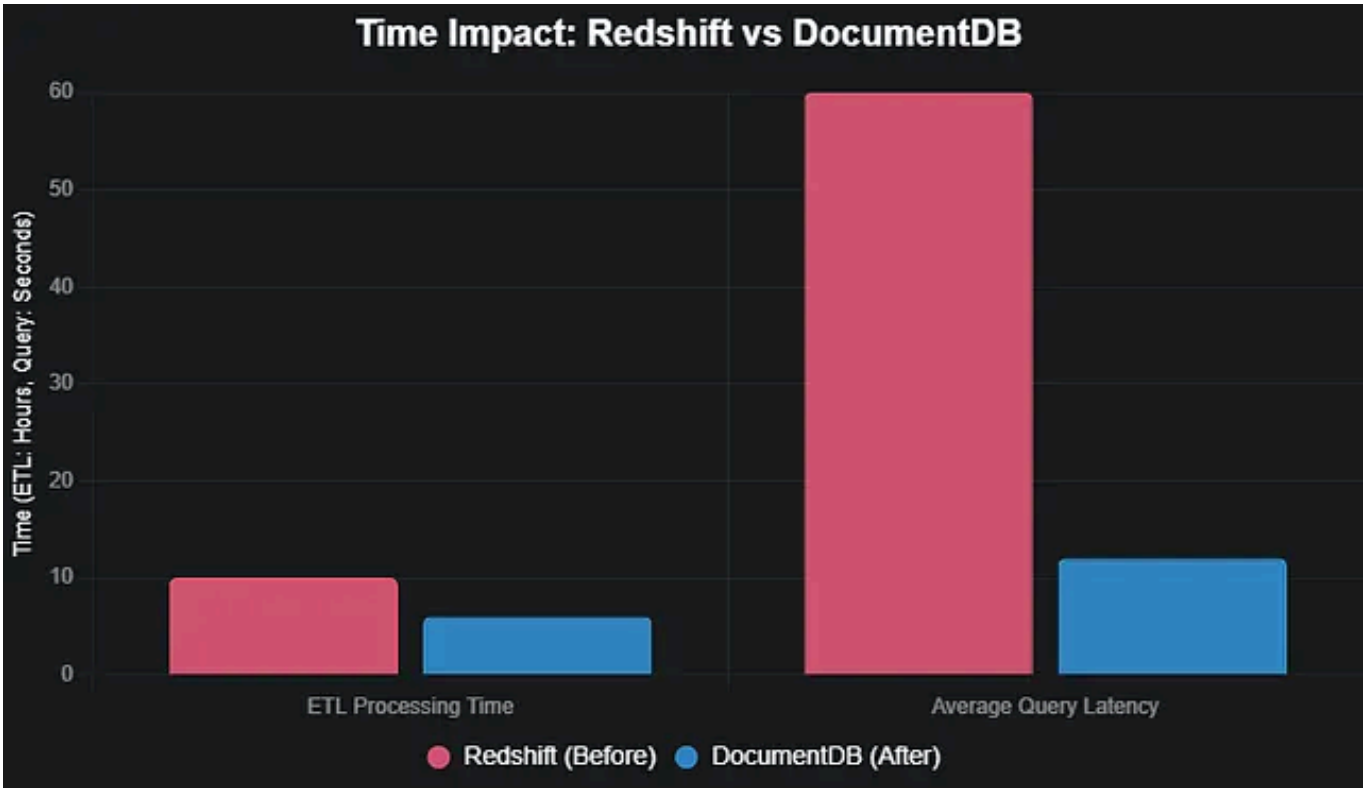
This pipeline turned our system from batch-only to event-driven, boosting user satisfaction.

```
{
  "_id": "d41d8cd98f00b204e9800998ecf8427e",  // MD5 of business_key
  "business_key": "12345",
  "data": {
    "name": "John Doe",
    "department": "Engineering",
    "salary": 85000,
    "hire_date": "2020-05-15",
    "status": "Active",
    "location": "New York",
    "email": "john.doe@example.com",
    "title": "Senior Engineer",
    "skills": ["Python", "AWS", "Data Pipelines"]
  },
  "last_updated": "2025-09-03T12:00:00Z",
  "audit": {
    "created_by": "system_batch",
```

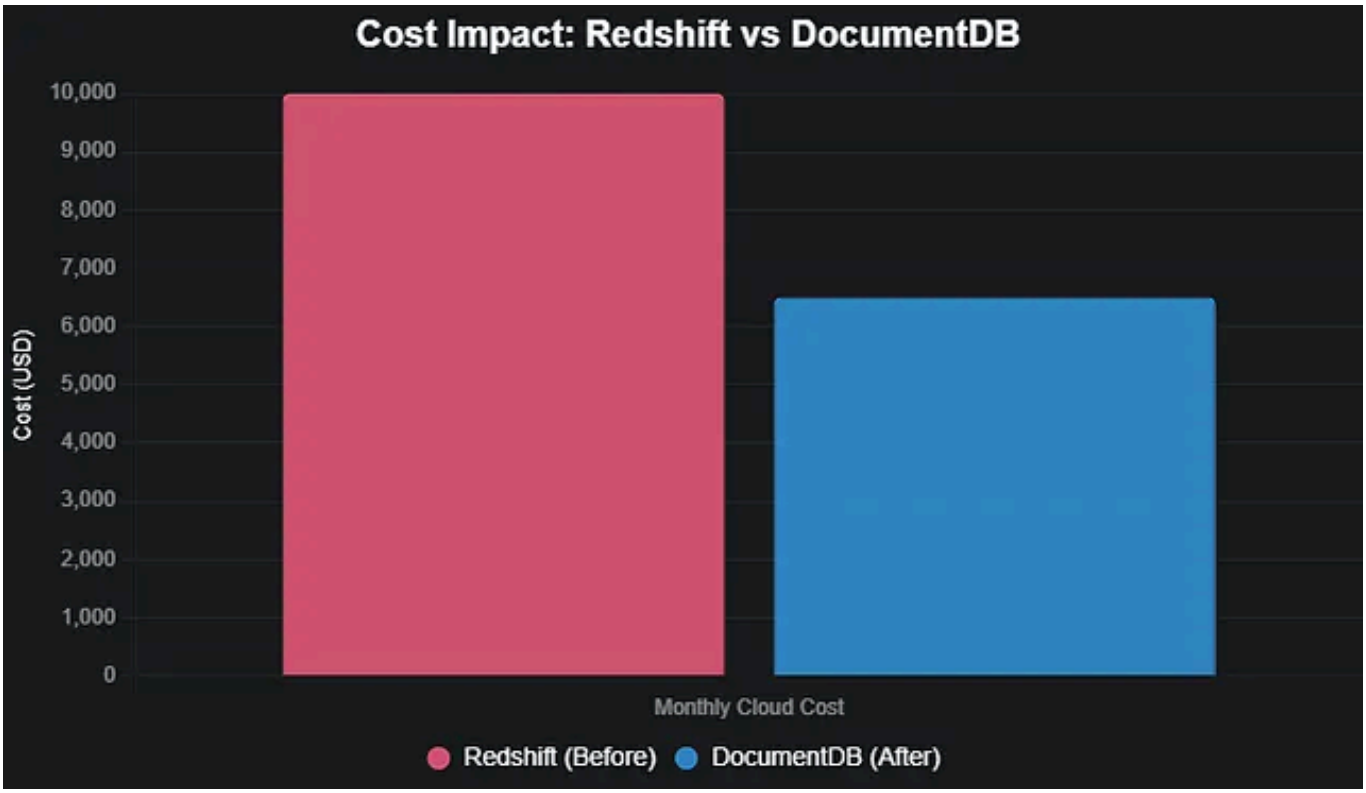
```
      "version": 2
    }
  }
```

We evolved from tilde-separated strings to nested JSON for easier querying with MongoDB's aggregation framework.

Before-and-After Impact on Time (Query/ETL Latencies)



Before-and-After Impact on Cloud Costs (Monthly Billing)



Lessons Learned and Future-Proofing Our Data Ecosystem

The migration cut ETL times by 40%, query latencies by 80%, and costs by 35% through DocumentDB's efficient storage and our lean designs.

Key takeaways: Involve stakeholders early for buy-in, automate testing with tools like dbt, and monitor everything — we used Prometheus for metrics.

We learned that hybrid models ease transitions, and pre-computing pays dividends.

Data Engineering | Data Engineer | Databricks | AWS | Migration