How we used the AWS Cloud to optimize a MongoDB-based document management system

17 September 2021 - 5 min. read

These days a lot of applications are developed with the cloud in mind: an excellent architecture isn't the only thing we need to consider to succeed.

In today's article, we'll see that it's always better to stop and choose the right solution, even if a technology is available both on the cloud and on-prem.

Storage is a perfect example to state our case: there's a wide choice of available services that can help you reach your goal: using the right one can be a game-changer, enabling you to reduce maintenance tasks and lower operational costs.

If you take your time to explore cloud-native technologies you'll find better solutions, even if using them can lead to an application refactor.

We'll tell you a case that involved data migration, even if the application was already running in the AWS Cloud.

Some time ago, we were tasked to help improve performance and reduce costs of a custom-developed document management system running on AWS that could be hosted on-prem and also on the cloud.

The application was designed for high availability and scalability, with a microservices architecture and using a MongoDB cluster to store documents as BSON (https://bsonspec.org/).

A huge amount of documents was expected to be ingested, so elasticity was the key factor that led to cloud adoption.

After porting everything in the AWS Cloud, everything went fine: an ECS Fargate cluster was used to run containers with autoscaling enabled, using a MongoDB Atlas cluster with three nodes to store data.

After some time (and millions of documents later), issues began to arise, in the form of a noticeable increase in the monthly AWS bill for storage occupation and availability zone traffic.

Data transfer costs were due to cluster synchronization and traffic: instances holding data were deployed in different AZs to maintain high availability, so charges reflected the amount of traffic the cluster had to sustain (https://docs.atlas.mongodb.com/billing/data-transfer-costs/).

Storage costs are proportional to the size of EBS volumes required to store data and MongoDB's "replicationSpecs" parameter.

In addition to the aforementioned costs two additional nodes had to be added to the cluster to maintain a high level of performance during traffic spikesWhen a node needed maintenance or, for worse, failed, additional work was required to keep up with the service level agreement.

It was becoming clear that something had to be done to lower costs and the amount of maintenance required.

Sometimes the best solution to a complex problem is the simplest one: meet Amazon S3 (Simple Storage Service), one of the first services generally available on the AWS Cloud, released in March 2006.

S3 is an Object Storage designed for performance, scalability, availability, and durability (with its famous 11 9's). It has a wide range of cost-effective storage classes and data management APIs that have become a de-facto industry standard. https://aws.amazon.com/s3/

Some of the main reasons we chose Amazon S3 as a storage service are

- Lower cost/GB ratio (compared to EBS)

- Out-of the box multi-AZ data replication and encryption

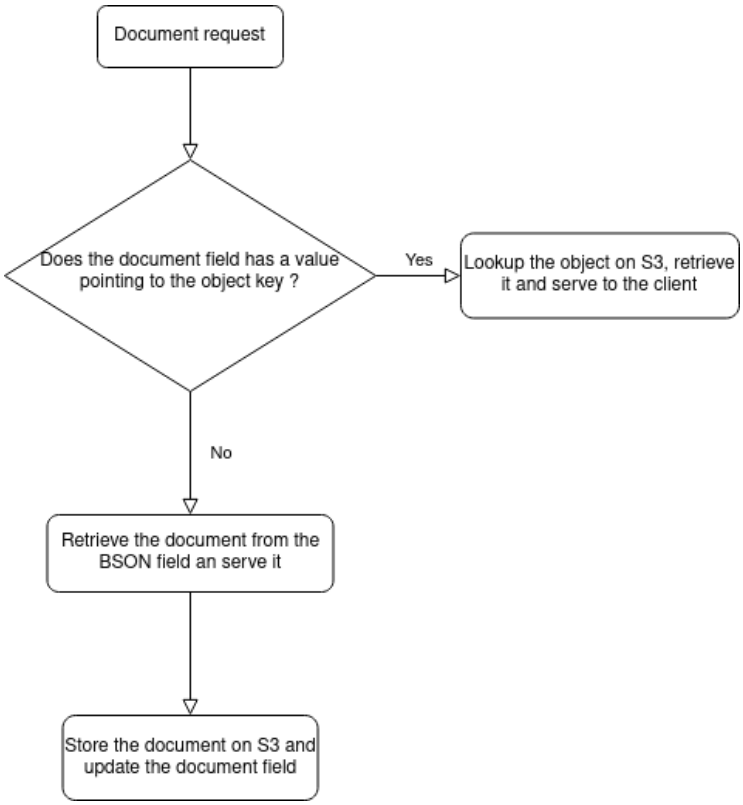
- Intelligent tiering helps to categorize data based on access patterns, keeping costs low.
- Bucket Cross-Region replication is easy to set up and helps with disaster recovery scenarios

After a quick proof of concept, the application was ready to use S3 to store user's documents, but data migration was needed, with the shortest reasonable downtime.

Taking the entire application offline, running a script to migrate data, and deploying a new release was too risky and not feasible since the amount of data to migrate was too big to handle, even during the night.

We needed to rethink our approach and shift the migration on the application side. After some design and brainstorming sessions, we came up with a safe plan that didn't require downtime; in the end a small application refactor was needed to implement a new logic for document retrieval.

At first, a new field representing the object key in a bucket was added to the schema, then the document retrieval logic followed this flow chart:



This logic worked well, but our need was to complete the data migration in progress to resize the MongoDB cluster, reduce costs, and ensure that all the documents were stored on S3.

We developed a custom python script to leverage the application logic to migrate the documents: requesting a document was enough to start the object migration.

We started with datasets containing the list of documents to migrate: a single dataset contained about 70 million documents, so we had to split them in smaller batches.

To store the migration status for each object in a batch we needed a high performance database that could handle millions of key-value records: our choice was (obviously) a DynamoDB table.

We used the object id as table key, saving the migration status and a hash of the retrieved object.

We chose to run the script in parallel on EC2: after a quick run on a t3 family instance we saw that we needed more memory and multiple m5 instances that could execute multiple parallel migration jobs.

To handle script failures we implemented a logic to check the migration status on the DynamoDB table before making the request to the application to avoid unnecessary calls.

An additional script was developed to check hashes of successfully migrated objects on S3 against the saved entry on DynamoDB, to be sure that no data corruption could happen.

The migration took a month. 40 Terabytes of data were successfully moved, without impacting users and availability, even during traffic peaks.

A significant reduction in costs and maintenance allowed the business to take off and expand the user base. Costs were lowered and performance improved

Takeaways

The Cloud isn't a silver bullet that magically solves every problem. We need to rethink applications (or small portions of them) to use it efficiently, even if they seem to run flawlessly when ported in a cloud environment.

When products scale, they need to adapt to changes to survive and grow; complex solutions aren't often the best choice: sometimes the simpler is the better.

Damiano Giorgi

Ex on-prem systems engineer, lazy and prone to automating boring tasks. In constant search of technological innovations and new exciting things to experience. And that's why I love Cloud Computing! At this moment, the only "hardware" I regularly dedicate myself to is that my bass; if you can't find me in the office or in the band room try at the pub or at some airport, then!
