Tuning and Scaling a Data Migration on AWS

Morten Heine Sørensen
mhs@formalit.dk

ABSTRACT

Every system is eventually replaced by another system. When this happens, we often migrate the data from the old system to the new one. This data migration is usually achieved by some *migration application*. Characteristics of it often include:

1. The computing resources are needed for a short time.
2. It is required to complete within a specified time frame, during which the old and new system are closed down.
3. The migration of each entity requires computationally significant validation and transformation.
4. The code will be used for a short time and discarded.

The first point makes cloud computing ideal for migration because we can access and pay for computing resources for a short period of time. In fact, so does the second and third, as we may need to scale resources aggressively to finish the process in the required time frame. The last point entails that investments in the code (e.g. tuning) will have a short time to yield returns, so we instead look for brute-force short-term aggressive scaling, again emphasizing the use of cloud computing.

This paper presents a real-life performance optimization of a migration application dealing with a customer management system handling a million customers. Both the new system and the migration application are running on AWS.

The total duration of the most critical step went from 9 days in the initial version down to just over 1 hour. The optimizations included tuning as well as both horizontal and vertical scaling.

The paper distills the optimizations into general techniques that can be used by most migrations.

Categories and Subject Descriptors

D.2.8 **[Software Engineering]** Metrics – Performance measures.

General Terms

Measurement, Performance.

Keywords

Cloud, AWS, EC2, RDS, EFS, SQS, performance, data migration.

INTRODUCTION

Company X delivers a certain kind of supply to private customers and has acquired Company Y, which has developed a customer management system. The system keeps track of the customer's consumption, calculates the customer's bills, sends invoices to the customer, and handles the payments from the customer. The system has integrations to external systems for receiving consumption data, receiving payments and maintaining customer data. Company X wishes to replace its existing system with the system developed by Company Y.

The new system looks as outlined in Figure 1. It has an application server (App) running the application used by customer service. It is also used by the customers via a self-service web application and a mobile app.

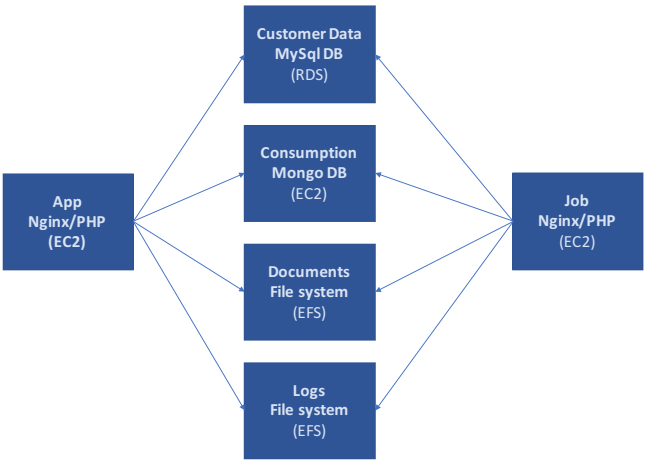

Figure 1. The new system.

The application server uses a database server to store customer data and another server running Mongo DB to store consumption. A batch server (Job) is running all jobs. The application and batch server share a file system where the system keeps application logs as well as documents, which are mainly letters and invoices stored as PDFs.

The migration application runs on the batch server as just another set of jobs and was written in PHP like the system. This has the advantage that the migration application can make use of existing libraries, validation rules, event handlers, etc.

The migration runs in these steps:

1. Import data.
2. Filter data.
3. Clean data.
4. Migrate reference data.
5. Migrate customer data - part I.
6. Queue customer data.
7. Migrate customer data - part II.
8. Reconciliation customer data.
9. Migrate documents.
10. Reconcile documents.
11. Synchronize data with external integrations.

Prior to Step 1, there are many other steps which close down the old and new system and export data from the old system into a set of files, used in Step 1. Similarly, after step 11, there are steps where systems are brought back up.

As the migration is done in chunks of 250.000 customers over a number of weekends, the old system continues in production until the last chunk is completed (in practice, the first two runs migrated 500 and 30.000, respectively). When the first migration ran, there were 5.000 existing customers in the new system, which had been created directly as new customers in the system.

When the migration was functionally more or less complete, we started looking at performance. To this end we needed to understand just how fast the migration had to be.

As mentioned, the 11 steps were part of an overall process with many other steps. Along the way it was decided that a total time of 6 hours for the most critical step during a migration weekend would be sufficient. It was realized from the beginning that some steps could run outside the closing period, but the most critical steps should preferably run in the allotted hours. Thus, our optimization exercise began.

1. IMPORT DATA

Step 1 of the migration is to read 18 large files (contained in a zip file) exported from the old system and insert their lines as rows, file by file, line by line, directly into corresponding staging tables created in the database containing the new system's data.

The elements of each line in the file map directly to the columns of the corresponding table. In principle, these tables directly reflect the data model of the old system. In practice, they were already partly transformed towards the model of the new system. Nevertheless, significant transformation, validation, etc. remained and were the job of Steps 4-7 of the migration.

Reading the files line by line turned out to be rather inefficient and took around 10 hours for 250.000 customers. Due to the identical format of the files and tables, we decided to instead use mySql's import feature, which can directly import an entire file into the corresponding table. Running the imports sequentially brought the time from 10h down to around 1h.

Technique #1. *Use imports instead of inserts for large sets.*

The files were read in a certain sequence due to foreign key constraints. However, we temporarily disabled all constraints during the import and imported the 18 different files in parallel. Now the largest of the files became the dominant factor in the consumed time, which went further down to 15 minutes.

Technique #2. *Run similar, independent tasks in parallel.*

The 15 minutes time were divided between download of the files, actual import, and a post processing phase that encrypted clear-text passwords. Each took around 5 minutes.

The overall migration process was overseen by a migration master directing when each activity was to take place. When we got the files on an SFTP-server, we had to wait until the correctness of the data in the files had been validated by people from the old system, before we could load them into tables. However, we could make use of that waiting time by splitting the import into an explicit download step, which we could perform ahead of time, and then the actual import and post processing, when granted permission. This cut the time down to 10 minutes.

Technique #3. *Run tasks (e.g. downloads) ahead of time.*

At this point we decided no further optimization was relevant.

2. FILTER DATA

Once the exported files were imported into the staging tables, the people responsible for the old system would further validate the correctness of the data. Some agreements would be deemed inappropriate for migration, despite extraction into the files.

We were sent a short list of such agreements, and Step 2 consisted in reading the file and inserting the lines into a table. During migration, all agreements mentioned in this table will be ignored.

This step is essentially just another import and needed no optimization as it was very quick from the beginning (less than 1 minute).

3. CLEAN DATA

When migrating data from an old system, there is often a problem that the old system's data contains garbage of various kinds. It is desirable and often necessary to clean up the data. This can be done in the old system, in the new system, or in the migration.

In the present case, mainly one kind of cleaning was done in the migration, which was to collapse different customers that were actually the same. This happens more often than one would think in legacy systems. In this particular case, the business representatives had created a mapping indicating that customer A's data should be merged into customer B.

The migration handled this by reading the file with the mapping and creating a corresponding staging table with the data, and then changing all foreign keys pointing to customer A to point to customer B instead.

In reality, the process was substantially more complicated and resulted in many difficulties; but from a performance perspective there were no major challenges. Depending on the size of the mapping (which went from very big to very small) the time varied between 15 minutes and 2 minutes.

At one point, when the file was particular large, the migration terminated with an out of memory error, because the entire mapping was read into memory. We therefore increased the memory available to the process from 1Gb to 8Gb. After this, we did not experience the problem again.

Technique #4. Monitor memory consumption and make sure you have enough.

4. MIGRATE REFERENCE DATA

Step 4 consisted in migrating reference data. These were mainly products and product prices.

It is natural to split migration of customer data and reference data, but there are also some tangible advantages which will be clear in Steps 6-7.

In Step 4-5, the contents of the staging tables that contained reference data were transformed to the new system's model,

and it was checked that the data did not already exist in the new system. In case of partial overlaps (for instance, in the duration for a price), errors were reported. As the amount of data was quite small, this step was quick from the beginning, around 2 minutes.

Technique #5. *Split reference data and customer data.*

5. MIGRATE CUSTOMER DATA PART I

Customer data was divided into consumption and everything else. The consumption came in a file that almost matched the new systems model in Mongo DB. We devised a couple of sed-scripts for the minimal pre-processing needed and then used Mongo's import feature to import the consumption files directly.

There could be some 8-10 files of varying sizes and they were imported sequentially. We did not study Mongo's import in enough detail to conclude whether parallel import would work. The total time of the sequential imports was around 1 hour, which we decided to live with.

6. QUEUE CUSTOMER DATA

The remainder of the customer data consisted of agreements, customers, etc.

In Step 6 we place every agreement on an SQS queue.

In step 7, a separate process takes an agreement from the queue, reads all data from the staging tables, validates it, transforms it and inserts it into the new system's database.

There are several reasons why using a queue is a good idea in this case. For instance, if the process breaks down we can easily resume it, and progress monitoring is easy. There are also some performance benefits, as will be shown below.

This step took around half an hour, but we decided not to improve it, for reasons explained in the next section.

Technique #6. *Migrate large number of entities over a queue.*

7. MIGRATE CUSTOMER DATA PART II

The level of granularity in Step 6 was not customer, but agreement. Why? Well, a customer may have more than one agreement, some of which may fail migration, while others may succeed. Thus, it was desirable to treat each agreement as a unit.

The migration of agreements and associated entities was the heaviest process in the migration (not counting documents), initially spending some 3 seconds for every customer, or a through-put of 20/min. With 250.000 customers that adds up to almost 9 days. This was on an RDS DB server, while the job server was an EC2 (exact instance types deliberately omitted).

This timing was obtained after adding all the indexes we thought relevant on the staging tables, but without any other tuning.

Our first idea was to run 10 parallel workers, reading from the queue and inserting into the new system. Initially this resulted in some errors, because a customer may have several agreements. Two workers would simultaneously check whether the customer of an agreement had already been created and decide that it was not the case. When they both tried to create the customer, one or the other got a duplicate key exception on the customer number.

To solve this problem, we could have implemented a locking mechanism, but with many threads reading and inserting simultaneously we would rather try to eliminate the need for locking than risk introducing new bottlenecks. Therefore, we instead implemented a sharding mechanism. Instead of one queue with 10 workers we shifted to 10 queues (numbered 0 to 9) with each one worker. An agreement would be sent to the queue corresponding to the last digit of the customer number. In particular, all agreements for the same customer were sent to the same queue, and since it had only one worker, they were processed in sequence. This increased the through-put to 120/min, i.e. a factor 6.

Technique #7. *Use sharding to avoid parallel workers that create or update the same rows.*

Strictly speaking, we could risk other kinds of shared data than customers so we should check our model and verify that there is no problem. Figure 4 shows the model of the data exported from the old system (the model for the new system is sufficiently similar that the reasoning carries over). Analyzing the model (and understanding the precise semantics of the arrows) we see that, starting out from two different agreements, the only entity types with an entity that can be reached from both agreements are customers (and their logs and addresses). The only exception is reference data, but this is already migrated before agreements. Hence our sharding is sufficient.

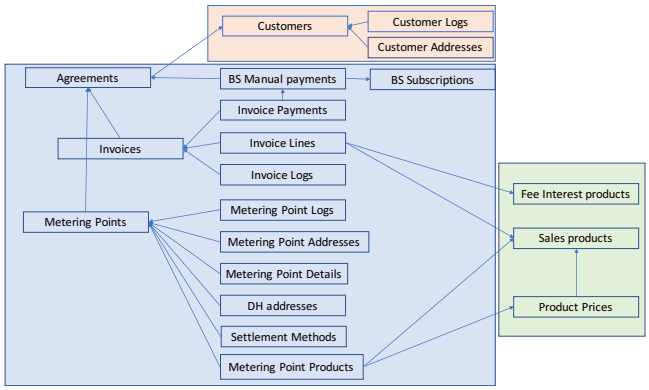

Figure 4. System model.

An alternative to the sharding would be to migrate customers and their logs and addresses before agreements. (The ultimate philosophy in this direction would be to migrate all entities type by type in some order respecting the foreign key constraints, or to temporarily disregard the constraints, as we did for the import.) However, this would interfere with our transaction scope. That is, if all agreements of a given customer failed, we would have to afterwards explicitly delete the customer from the new system again. Besides, in many cases the exact migration of an entity depends on the details of its related entities, which makes the transactional approach more desirable in the first place.

Returning to our improvements, increasing the number of workers to 200 (using modulus of the customer number to identify the relevant queue as a generalization of the last-digit approach), did not increase the through-put. The job server and DB server both ran on around 10-20% CPU, so computing resources was not the problem. We had some bottleneck.

There was a hypothesis that the queue-mechanism itself was inefficient. It would be laborious to replace it. So before doing so, we made a simple experiment with a worker that did nothing else than fetch from the queue. It scaled perfectly from 10 to 100 workers.

Technique #8. *Make sure you have a problem, before fixing it.*

We used New Relic to show where in the code most time was spent. It showed that significant time was spent writing to files. In fact, the workers all wrote significant amounts of data to the same single log file to record what was going on during the migration. That very big log file was residing on EFS giving some delay plus the 200 workers were blocking each other.

Most of the information was available from some control tables that were updated during migration, so most of the data in the log file was not strictly necessary. Removing the writing to files except in cases of errors and using 200 workers increased through-put by a factor 4 to around 500/min. Now the DB server was running 100%. (Later, when an error had been introduced in the migration code leading most agreements to fail, the remaining file writing caused the migration to come to a full stop before it was complete, due again to the locking.)

Technique #9. *Don't write intensively to shared files from many workers.*

Technique #10. *Use New Relic to find inefficient code.*

Since the DB server was now fully exploited we started looking at tuning options. New Relic again gave some hints on where the time was spent. Also, we used mySql's slow_query log to find inefficient SQL, in particular missing indices.

Technique #11. *Use mySql slow_query log to find inefficient SQL.*

We added two new indexes to the new system, one on the Countries table and one on the Products table. The Countries table became populated with all known countries during the migration (before migration it held just one hardwired country for all customers), so it had not previously been a problem. That improved through-put by 25%.

Technique #12. *Make sure you have the needed indices in the new system, e.g. for reference data.*

We moved some synchronization with external integrations to after the migration, which improved through-put by 44%.

Technique #13. *Handle non-critical migration as post migration.*

During migration of an agreement, we inserted a lot of data, and some this data was read again repeated times during the remainder of the migration of the agreement. For instance, the customer was read when saving invoice lines. This approach was due to a mixture of different things including how the model layer of the application worked (need to have object rather than just key). We changed this so that almost all data created during migration of an agreement was cached until the processing of that agreement was done. That improved through-put by 50%.

Technique #14. Cache Customer data for the duration of migration of the customer/agreement.

The type of entity with the highest number was invoice lines (around 20 for each agreement on average). Every invoice line refers to a product. However not all of these products were migrated to the new system. Hence the migration first looked for a migrated copy of the product from the old system. If not found, it looked for a fee (a special kind of product) in the new system. If still not found it used a generic translation scheme based on naming of the old product to map to a product in the new system. This required a number of database calls for many invoice lines. Instead we read the full list of products and fees in the old and new system into memory once and for all in each worker.

Technique #15. Cache reference data for duration of worker.

Adding all these tuning improvements together in practice gave a factor 2 in through-put. With 200 workers, we thu arrived at around 1000/min.

This was in a test environment. The production environment was a bit slower. We removed New Relic from production and believe it went a bit faster.

Technique #16. *Disable performance monitoring in production to avoid overhead.*

For those new tables were we only insert, not read, we could consider deleting the indices and recreating them after the migration, as maintaining indices supposedly can be expensive. We did not do it, however, as it was not badly needed and we were not sure how much would be gained.

Technique #17. *Estimate benefit before making changes.*

Recall that the step of populating all the queues took around 30 minutes. Initially we did it before starting the workers. However, we changed the order so that we started the workers ahead of time with nothing to do. When the messages got placed, migration thus started immediately cutting 30 minutes from the overall time.

Technique #18. *Start queue workers before placing messages*.

The job server was now running on 20% CPU while the Database server was running close to 100%, see Figure 2-3.

Figure 2. DB server CPU load.

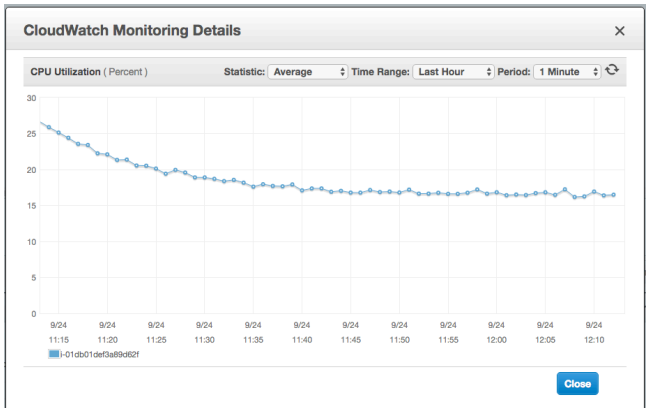

Figure 3. Batch server CPU load.

We tried replacing the RDS by a 4 times larger model, reasoning that we needed a factor 4 due to the difference between the batch server and DB server. In fact, this gave approximately a factor 4 to around 4000/min.

Along the way, the through-put would decrease to around 3700 and then come up to 4000 at the end. We tried various experiments, but were not able to change this pattern.

Technique #19. *If Server A is calling server B, and server A has less CPU load than Server B, try adding more juice to server B.*

We also tried with an even larger DB server. The through-put increased slightly compared to the previous instance type, but not only around 25% and the CPU load went curiously up and down along the way, which drove us back to previous instance type.

Technique #20. *Experiment with larger servers. You can always wipe them again, if they don't help.*

Similarly, we use small test environments and only increase them when we run big test migrations. Also, the production Job server is increased during production migrations.

Technique #21. *Use small servers, and then only increase them when needed.*

8. RECONCILIATION

Reconciliation is a central part of our migration, as it should be in most cases. It has some impact on performance too.

In each migration run, we get a number of entities of each type (i.e. customers, agreements, invoices, etc.) from the old system. For instance, we get 250.000 agreements. We refer to the number of imported entities of type E as $IMPORTED_E$.

As we migrate all the entities, some may fail. For instance, an invoice line may refer to a product that cannot be identified by any of the available means. The number of failed entities of type *E* is $FAILED_E$.

As discussed earlier, some entities are excluded. At the highest level, agreements may be excluded, and all the associated entities are then excluded as well. The only exception are those entities not linked to a single agreement. For instance, a customer may have multiple agreements. For a customer with one or more excluded agreements, the customer itself and his addresses and logs are only counted as excluded if all the customer's agreements are excluded. The number of excluded entities of type E is $EXCLUDED_E$.

The entities that are neither excluded nor fail are referred to as digested. The number of digested entities of type *E* is $DIGESTED_E$.

There are two different reconciliations. The first one is, for every type *E*:

$$IMPORTED_E = DIGESTED_E + FAILED_E + EXCLUDED_E.$$

That an entity is digested does not necessarily mean that it is actually migrated, because it may be filtered. For instance, some imported customers may already exist from a previous migration run. Also, filtering may happen due to differences in model between the old and new system. For instance, in our case, some customers have multiple products simultaneously in the old system, but only one product in the new system. That is, the relation to all products except one is filtered. We use $FILTERED_E$ to denote the number of filtered entities of type *E*.

There is a second reconciliation dealing, among others, with the filtered entities. To explain it we need some additional terms.

Before and after migration, we count the number of entities of type E, resulting in $BEFORE_E$ and $AFTER_E$. In addition to this, we count the number of entities of type *E* that we add independently of any entities we receive of that type from the old system. For instance, there is a customer log for each customer with entries for significant things that have happened to that customer in the old system. These are migrated to the new system. But in the migration, we furthermore add one additional entry to explain that the customer was migrated. The number of added entities of type *E* is denoted $ADDED_E$.

The second reconciliation states that

$$AFTER_E - BEFORE_E - ADDED_E = DIGESTED_E - FILTERED_E.$$

In other words, the number of entities that have been created in the migration, except those that did not result from any imported entities, must match the number of digested entities, except those that were filtered.

It is surprisingly difficult to get the two reconciliations right in all cases.

Each of the staging tables has certain fields that are used to document the outcome of a migration run. These include:

- Excluded At
- Failed At
- Digested At
- Filtered At
- New Key

The first three indicate the time at which the entity reached its status and are mutually exclusive. Filtered At indicates the time a digested entity was decided to be filtered, if this happens. The last one indicates the key of the corresponding entity in the new

system, if the entity was inserted by the migration (so the entity was digested, not filtered).

Every entity created in the new system is indicated in a table called NewEntities with the following properties:

- run_id: a unique id identifying the migration run.
- record_type: E.g. Customer.
- record_id: the primary key of the entity in the new system (when the key is a number).
- added: expresses whether the entity was added or migrated from an entity in the old system.

This makes it easy to see, for a given entity in the new system, whether (and when) it was migrated. The Created_at field is generally not useful for this as that field is copied from the old system.

In terms of performance it is crucial to have indices on all the columns mentioned earlier in this section. Nevertheless, the reconciliation will involve a number of full table scans of tables in the new system, when we count before and after numbers. With the number of entities we have in our case, this was not a problem. The reconciliation ran in around 10 minutes, so we did not bother to optimize it.

However, when dealing with documents we had performance issues with the reconciliation, as will be discussed below

9. MIGRATE DOCUMENTS

As a separate part of the deliveries from the old system we received a number of zip files with letters and a number of zip files with invoices (typically around 10 of each). Each zip file contained thousands of PDFs and could be up to 10Gb.

The initial approach downloaded the files in sequence and then read directly from the zip files, one by one, and for each one put the file in the correct position in the folder structure and made updates to the database to link data to the file.

Initial measurements revealed this would take around 6 days. It turned out that around 25% of the time was download time, so we separated out the download in a separate process that could run ahead of time and use a separate thread for each zip file. The capacity of the SFTP-server (which we did not have under our control) appeared to be exhausted when we ran all threads simultaneously. Thus, we first downloaded the invoice zip files and then the letter zip files. This way the download time was brought down to 2 hours for letters and invoices each.

Similarly, we ran the actual processing of the zip files in parallel. This way the processing time was brought down to around 4 hours for all letters and invoices.

The total process was hence brought down to 8 hours, where only 4 was for the actual processing. As it was actually considered acceptable to complete the documents part of the migration after the closing period, we did not optimize further.

In one migration, we were close to running out of disk space, so we downloaded the zip files one by one and imported and deleted each one before downloading the next.

Technique #22. Monitor disk usage when dealing with documents and make sure in advance you have enough.

10. RECONCILE DOCUMENTS

The counting of documents before and after migration was done by actually counting files in the file system. While there were other ways we could count, looking at the actual files was the most certain way. However, as the number of files grew with the migration runs, the process became slow. After migrating 250.000 agreements, the counting took 10 hours.

To circumvent this, we traded some of the certainty for speed, and instead explicitly counted the number of files we added. That brought the time down to 5 minutes.

Technique #23. *Do not count documents in the file system.*

11. SYNCHRONIZE WITH INTEGRATIONS

One of the trickiest parts of the migration was that the new system exchanges data with external systems, both inbound and outbound, and large parts of the model must be synchronized locally and externally.

During normal operation, the number of local or external changes is relatively small, but after a migration, all the migrated entities are affected. This generated a lot of work for the system, but we quickly came to terms with this traffic by making some optimizations in the new system. These processes ran after the closing period, so spending some time was not a problem. However, some of the procedures had to completed within one day due to market regulations.

Technique #24. *Make sure the new system is optimized to handle the amount of data and users from the migration.*

12. TOOLS

The main tools used in the initiative were these:

- **New Relic.** This was used to identify methods in the code where the majority of time was used. It ran on test and production environments, until we switched it off to reduce risk of performance impact.
- **MySql Slow Query log.** This was used to identify slow queries and queries scanning many rows. It ran on test and production environments, until we switched it off to reduce risk of performance impact.
- **AWS resource monitoring**. This was used to monitor memory and CPU usage.
- Local tools. Our local development environment ran in a Docker container, where we made measurements when running with just one queue worker.

Technique #25. Use tools to measure performance and identify root causes.

13. CONCLUSION

We have distilled some 25 techniques or principles that we think can be used by most migrations. The below table summarizes the

achieved improvements based on the techniques. We did not make good measurements of the integration with external partners, so it is omitted from the table.

Step	Before	After	Factor
Import data	10h	15m	40
Filter data	1m	1m	1
Clean data	15m	15m	1
Migrate reference data	2m	2m	1
Migrate customer data - part I	1h	1h	1
Queue customer data	30m	30m (0m[1])	1
Migrate customer data - part II	9d	1h 30m	144
Reconciliation	10m	10m	1
Migrate documents	6d	8h (4h[2])	18
Documents reconciliation	10h	5m	120
Total	15d 21h 58m	11h 48m	32

Note 1: By starting workers before placing messages on queues.
Note 2: By downloading documents in advance.

Acknowledgements. I am indebted to the members of my migration team for cooperation on the migration code and optimizations. I'm also indebted to the application team and the infrastructure team for experiments with AWS. You know who you are.