scalac

best scala hAkkers!

Migration to Cloud

a case study

may 2019

Category: The project included both backend and frontend technologies.

- **The backend** was written in java, some small microservices in Python.
- **The frontend** was mainly written in HTML and JavaScript as static files.

Industry: The entire infrastructure for was designed for a private medical care system.

Challenge

The bare-metal servers were not running properly and our client was **wasting a lot of resources just trying to host their systems**. The infrastructure had been set up in the **pre-cloud era** and some of the elements of the system had been set up in **different hosting locations**. This meant it was hard to manage access, and transfer data.

The costs of server maintenance, getting new parts and keeping them all up and running at the same time was becoming more and more problematic and had reached the point at which it was directly influencing company stability.

Scaling up the infrastructure was also hard because of physical limitations and deprecated software, which did not support automatic scaling.

The client's team were frustrated and wanted to improve things but without serious changes in the approach towards the host systems, only small improvements could have been made. Unfortunately, the infrastructure needed much bigger changes.

This case study is a good illustration of how we deal with problems and deliver final solutions.

Solution

We investigated several different solutions. A private cloud was an option. However, this was rejected because private clouds are dedicated to a single client and this wasn't possible in this case.

The customer planned to create an infrastructure that could be used for multiple business targets in the future. After rejecting this option, a choice was made to go for AWS, mainly because of its leading position on the market.

The approach to "migrate" the client to a cloud had to take into account a few important risks:

- During migration, the system could go down for a **maximum of 60 seconds only**.
- There were **more than 90 different servers** with different operating systems (Microsoft and Linux) which had to be migrated in a strict order so as to keep availability high.
- The system was **separated physically and would have to be unified** in one solution supported by AWS.
- The **database infrastructure was suboptimal** and had to be redesigned before any migration could begin.

The process we designed was as follows:

1. **Dev and test accounts** were created in AWS to test progress of the migration.

2. In order to decide how the infrastructure should be designed, we had to **create an architectural design diagram**.

3. **A comparison of the different hardware** options was made (cloud vs bare-metal) to ensure the same farm of servers would be launched in AWS.

4. We started by **creating an infrastructure in AWS**:
   - Networking parts (VPC, security groups, routes).
   - EC2 fleet for services.
   - CloudFormation templates to automate server deployments.

- Auto Scaling Group for ensuring high availability of services.
- AWS RDS - database solution from AWS resources.
- Application Load Balancers above EC2 instances to ensure proper traffic division.
- S3 buckets to host static web content.
- Bastion host + NAT Gateway (to hide resources in private subnets from outside access).

5. **We selected a subset of servers and services** that could be easily migrated to AWS without harming the system's availability.

6. The migration of less important services took around 2 weeks and downtime during this time was possible and could easily have happened. We were migrating services mostly based on RHEL + databases (mostly mysql and postgresql).

7. **While the less important services** were being migrated, we started preparing the more crucial ones.

   We spent most of our time working on this step (almost 2 months). With the client's acceptance of downtimes of up to 60 seconds per service, we cloned the whole prod environment to our pre-prod to check if such max downtimes were possible.

   We decided to create a Disaster Recovery strategy by using CloudFormation templates in different Availability Zone in the case of any future failures. Our decision to use ClodFormation to raise problems with important services in the case of any failures reassured the customer.

   The biggest problem regarding migration of the production environment was to ensure fast domains resolved to the proper IP addresses. To ensure this worked as fast as possible, we moved the domains to AWS (Route53) which resolves records efficiently.

8. **We used docker containers** when possible to ensure portability between servers. Docker images were pushed to the DockerHub repository. They were also used for dev and prod services.

9. One of the breakthroughs was to create **Auto Scaling Group + Scaling Policies**.

This ensures that if CPU usage goes above a specific threshold, it adds another EC2 instance to balance the load and prevents any application slowdown (or even excludes any downtime). This reassured the customer that despite any high loads, the services would still work.

10. To keep the team updated, **Zabbix was implemented as a monitoring tool**. We also used CloudWatch (an AWS monitoring tool) to receive alerts if any service went down.

11. **A CI / CD strategy was worked** out to make new releases deploy automatically. We used Jenkins and the GitHub repository for this, but not all of the services were deployed this way. Only the most important and developing ones.

12. **The final step in the migration process** was to recover everything in the case of any failures. We set up another account to relaunch the CloudFormation scripts, to check if the Disaster Recovery worked. We shut down services deliberately to check if this infrastructure was self-healing.

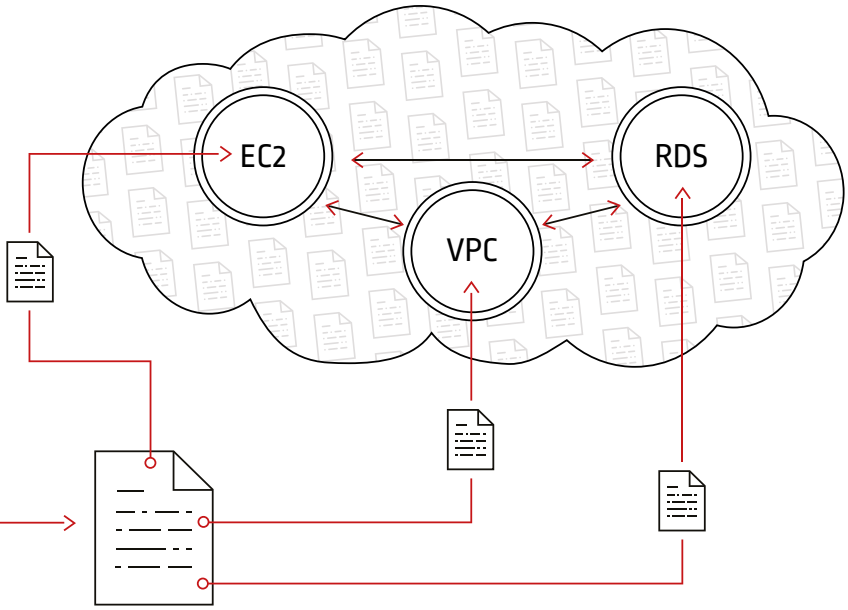
Technologies and tools used during the whole process:

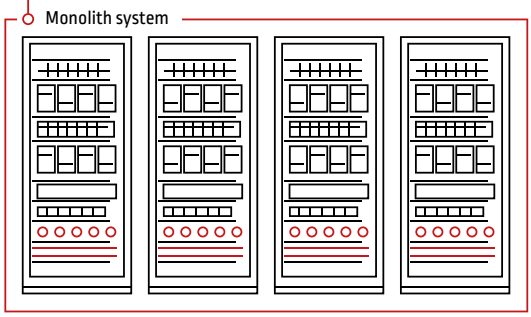
The final outcome was to enclose everything in AWS. To do this, we tried to use all necessary resources available within AWS. On top of the AWS resources, we also used: Docker images, Zabbix, OpenVPN to setup the VPN connections between instances and Jenkins for CI / CD to automate deployments.

scalac
best scala hAkkers!

Migration to Cloud

- Single location
- 40% cost reduction
- Self-healing & traffic-proof





- 90 diffrent servers
- 60s migration window
- Physically separated system

Results

Thanks to the auto-scaling features, our client **was positively surprised** by the way their infrastructure's load was handled.

In addition, after the migration to AWS, **the customer noticed significant reductions in costs**. He is now able to **control expenditure from just one location** (using AWS Billing). **Managing the architecture configuration is much easier** having it all in one place, without having to constantly jump from one multiple data center to other centres. The whole process took around 3 months, which included planning, testing and implementation of the changes.

However, **the greatest advantage** of having the infrastructure in AWS is that it is **self-healing**. Now, when one of the EC2 instances fails or goes down, mechanisms such as Auto Scaling Group and Launch Configuration can raise a new instance automatically without any manual involvement.

Also, thanks to the load balancers used in AWS, **services now have regular traffic** which means the **applications work consistently**. Using Load balancers helps to control the traffic on the servers and other virtual machines.

Aspects that have an impact on moving from bare-metals to AWS

1. Poor communication with staff in data centers,
2. Lack of devs and test servers in data centers to test solutions.
3. Problems with hardware in data centers (RAM corruption in server took 2+ days to replace).

Outcomes

What were the benefits the customer enjoyed?

- **A single location** where the whole infrastructure is held and managed.
- **Cost reductions** (of around 30-40%) with AWS.
- **Self-healing** and **traffic-proof** infrastructure.
- **One platform** for all billings and usage data.
- Cloud solution **elasticity**.
- **Up to 60% utilization of resources** (when a server is bought you pay for it but fluent changes are not possible, whereas in AWS they are).
- Around a **40% increase in service performance** (thanks to Load Balancers for example).

How did we cooperate with the customer's team?

1. The main team was based locally, but we worked with different teams from remote locations (Database teams) as well.
2. The Experts on our side and their specialisations - a few of our teams were involved: Database team, Networking team, Cloud engineers, DevOps and testers.
3. Project methodology: Agile.
4. How did we communicate to make the work go forward? Mostly on Skype for business, Slack (internal DevOps and CloudOps workspace) + emails.