The Copenhagen Initiative

How we migrated
the 6play VOD platform to
The Cloud, on AWS & Kubernetes.

Pascal MARTIN

Lead DevOps @ Bedrock, AWS Container Hero
@pascal_martin

The Copenhagen Initiative

How we migrated the 6play VOD platform to The Cloud, on AWS & Kubernetes.

Pascal MARTIN

This book is for sale at http://leanpub.com/tci

This version was published on 2021-06-22

Contents

About this sample

This is a sample of my book *The Copenhagen Initiative*, which you can buy in full version at the following address: leanpub.com/tci[1]

This book is available in PDF, EPUB and MOBI electronic formats.

What does this sample contain?

This sample reproduces some parts of the complete book:

- the first chapter: *Introduction*;
- the second chapter: *Our platform, our project*;
- the third chapter: *Discovering The Cloud and Kubernetes*;
- and a special bonus: the timeline of our migration to *The Cloud*!

I hope these few dozen pages will make you want to read more, while allowing you to check that the complete book will meet your expectations!

What does (or will) the book contain?

Most of the book has already been translated from French to English and you will get the following chapters right away when buying it:

- **Introduction**: why this book?
- **Our platform**, **our project**: an overview of our platform and applications, our technical background and our migration project.
- **Discovering the Cloud and Kubernetes**: why are we migrating to *The Cloud* and which provider are we choosing? How do we work with containers and what issues will an orchestrator solve? What was our first migration plan?
- **The Copenhagen Initiative**: our YOLO idea to quickly gain experience on an application deployed in production.
- **Our AWS setup**: accounts, regions and rights management. *Infrastructure as Code* with Terraform.
- **Our Kubernetes setup**: how do we manage our clusters, with kops, and what additional components do we install to make them fully functional?

[1] https://leanpub.com/tci

- **A first migration**: we are finally migrating our first application, with a minimalist deployment chain and a safe approach.
- **The beginning of the problems**: with an application in production, we finally encounter a first set of problems and we will present the solutions we have developed.
- **A stabilization phase**: what improvements have we made to our hosting, how do we manage monitoring, alerting and logging? In short, how have we evolved towards truly prod-ready hosting?
- **Cloud Native Cloud**: what impact does *The Cloud* (Kubernetes, managed services…) have on our projects and our teams?
- **Migrating other applications**: what choices have we made to migrate other more complex applications? What problems did we encounter and how did we solve them?

The last chapters have not been written yet (not even in French). They will be published and translated when I'm done with them, which may be in many months:

- **A second stabilization phase**: with almost all our applications deployed in *The Cloud*, we encountered another set of problems. And we made a lot of improvements to our new hosting!
- **CI, CDs and previews**: how does continuous integration work for containers? How do we deploy our applications painlessly?
- **Consumption/cost tracking**: the ability to launch any type of instance or service is very nice when we code… But after a while, let's look at the cost of our hosting and the optimizations we have put in place.
- **The development environment**: we quickly saw that this point was not going to be simple, because we use managed services and deploy containers to Kubernetes…

The published version of the book will of course be updated, free of charge, when these chapters are added or in case of corrections.

1. Introduction

This chapter was written in the summer of 2018.

1.1 Contents of this book

6play, the replay and VOD platform of M6 and other RTL Group channels, is hosted in *The Cloud*! Or rather, depending on when you read this, part of our platform is hosted in *The Cloud*.

Before 2018, our platform was exclusively hosted in a Parisian data center. There, we rented a room, racks, servers, network connections. When a disk broke or when we wanted to add RAM to a physical server[1], a technician would drive to the data center.
In 2018, we started our migration to *The Cloud*: we switched most of our hosting to AWS. We now use managed services when we can and our applications are often deployed to Kubernetes.

This book tells the story of this migration: how did we transform our hosting? What impact did it have on our projects? How did we organize ourselves? What choices did we make throughout the process? What did we learn, what did we change? And maybe even, one or two years later: what would we do differently if we had to do it all over again?
More than *"this is our platform, it is perfect"*, this book focuses on *"why"* and *"how"*.

I hope you will learn as you read and that I will help you answer questions you may already have.

1.2 Why?

If you are a developer or a system administrator, your work environment is changing. Rapidly.

Five years ago, large companies were buying servers, amortizing them over three years and hoping to use them for two more years. Today, for more and more companies, servers are a commodity. It no longer takes five weeks[2] to get a new one: you create three in the morning with one click and destroy them at the end of the day - or they automatically start when there's heavy load and shut themselves down as soon as they're no longer needed.
Hosting in *The Cloud* has matured well over the last five years and is still moving fast: Amazon, Google and Microsoft (to name a few) announce new services almost every day. Kubernetes, launched in 2014, took off between 2017 and 2018. And the CIOs got the message.

[1] Our applications were mostly deployed on virtual machines - but we also used a few physical machines here and there (and/or underneath the VMs).

[2] One week to get a quote at your vendor's site, a second week to get that quote validated by the purchasing manager, a third week to place an order and have the server delivered, a fourth week for a technician to go rack it in the data center and install the OS, and finally a fifth week to create the VMs it needs to host, configure them and deploy the applications. Add two more weeks if you need a server during vacation time. I'm exaggerating a bit, but not so much…

However, tools are not always as mature as we would like them to be. And new tools appear and disappear every day.
A few years ago, when I was a *backend* developer, I used to sometimes tease my colleagues from the *front* teams: *"hey, there's a new JS framework that came out, will you rewrite everything? Last time was six months ago ".*
Now, I have a *DevOps* role… And it's my fellow front-end developers who tease me at the coffee shop. *"So, is there another new Kubernetes tool out tonight?"*. Too often, my answer is *"I already have four tabs open in Firefox on this subject and I haven't finished unloading my RSS yet :-/"*.

I work at M6 Web, on the platform that powers 6play.fr[3]. We think we're doing some pretty interesting things, for ourselves and maybe for others - hopefully, for you, readers. In 2018, we also believe we are not behind when it comes to migrating to *The Cloud*, even if others are ahead of us. And we like to share our experience.
This book offers you more complete and longer content than a single blog post[4] and is more structured than a series of successive articles, without limiting ourselves to *"here is our perfect solution"* but sharing the steps we followed and our reasoning: the why and the how.

1.3 Our experience feedback

I'll talk about technique, I'll give examples of code[5], but above all, I want to share some feedback.
When we started our migration to *The Cloud*, we read articles and attended conferences. All too often, the information given corresponded to a *perfect state*: a platform designed from the start to be deployed in *The Cloud*, an application simpler than ours, articles that did not explain *"why"* nor present gradual steps. We also saw a lot of *hype*, before sometimes realizing by scratching a bit that the only brick deployed under Kubernetes was a POC[6] or a small non-business-critical component.
This was not our goal: we wanted to migrate to *The Cloud* and, therefore, go much further.
Here, I will as much as possible share the reasons behind our choices. We had many existing projects. We wanted to remove doubts. We had projects to deploy on fixed dates. Perhaps we invested time to understand better what we were doing? Or took shortcuts to move forward more quickly, knowing that we would then have to retrace our steps?

The writing of this book will be alternately technical and more romanticized[7], but its content will remain resolutely technical, while including what revolves around projects: a bit of politics, organization, management of priorities or budgets…

You'll soon realize we followed a *pragmatic* approach for our migration: we built *something that works*, even if we didn't always choose *the perfect* approach[8].
I will try to keep this *pragmatic* approach in this book. But, since I know how the story ends, I'll use

[3] http://6play.fr
[4] You can read our technical blog at tech.m6web.fr
[5] Some examples will be shortened or simplified in the body of the book, but they will still be heavily inspired by code used in production. Some examples will be abbreviated to make them easier to read or understand, and some will be reproduced in full in the appendix.
[6] Personally, I'm tired of *"I created my Kubernetes cluster on three Raspberry Pi"* - OK it's interesting to do and fun, but it's not *"prod-ready"* in the sense that we mean it here.
[7] Writing a tech book doesn't mean you can't have a little fun here and there ;-)
[8] Sometimes we were aware of it beforehand… Other times, a little less.

this hindsight to note here and there what went well or not, going from *"this idea was genius"* to *"oh well that was not so great afterall"*. It is this feedback that I wish to share.

1.4 Transparency and Confidentiality

We at M6 Web are used to being quite transparent on our technical stack. We talk about it regularly at conferences, we publish articles on our blog[9] and we open-source libraries and projects. This book is part of this willingness to share and open up: we hope to help our community move forward.

However, I cannot share absolutely all the information at my disposal. I won't talk in detail about *business-critical* or confidential information, such as:

- the exact numbers of visits to our applications or calls to our services and APIs;
- precise financial values;
- the terms of contracts with our suppliers and partners and information protected by an NDA[10].

I will therefore write *"this service receives about 10,000 requests per hour at peak time"* or *"its hosting costs us about 1000€ per month"* and not *"this service received 12,344 requests between 8pm and 9pm Tuesday 29th August"* or *"its hosting cost us 1234€ in August 2018"*. I will sometimes use fake names for projects or remain vague about their functional role: what interests me is how they are deployed and hosted.
I think this information, even if imprecise or given in orders of magnitude, will help you understand what I'm talking about and hope you won't hold it against me.

1.5 Writing Conventions

Throughout this book, I will follow a few writing conventions.

A passage formatted `like this` corresponds to online source code or a command. Lines formatted like this represent a block of source code or technical information:

```
<?php

echo "This is a sample source code ;-)";
```

To draw your attention to a trap, false or inaccurate information, or something you should not use in production, I will use a warning block formatted as follows:

This is a warning.

It draws your attention to a trap or information that I've reproduced but for which I insist: "don't put it into production as it is."

[9] https://tech.m6web.fr/
[10] NDA: A non-disclosure agreement.

Finally, to share additional information obtained in hindsight, perhaps to indicate that what I just said was a particularly good (or bad) idea, I will use a block formatted like this:

> **This is a "in retrospect" block**.
>
> I'll use it when I want to say "looking back a few months later, this is what we achieved".

1.6 About the author

My name is Pascal MARTIN.

I discovered programming when I was in high school, then PHP around the year 2000. I've been working in Web and PHP development for more than twelve years and I switched from a *"backend developer"* role to a *"DevOps"* role in 2017.
I joined M6 Web in October 2017, as lead of the DevOps team, to fluidify exchanges between the development and sysadmin teams, especially in the context of a migration of our hosting to *The Cloud*. This migration has been my main mission for a year now and will probably be for at least as much time.

> In 2020, the entity I was part of at M6 Web has been transformed into a separate company named Bedrock[a]. So, depending on when I wrote - or will write - each chapter, you might read "M6 Web" or "Bedrock". You can consider them to be the same company.
>
> [a] https://www.bedrockstreaming.com/

I am an AWS Container Hero[11] since 2020. I am the author of the book Développer une extension PHP 5 (FR)[12] and I am co-author of the book PHP 7 Avancé (FR)[13] (Eyrolles publishing). You can follow me on Twitter @pascal_martin[14] or email me at contact@pascal-martin.fr[15]. I also write on my blog: blog.pascal-martin.fr[16].

I'm of course open to any feedback, suggestions, advices and error reports you might want to give me ;-)

> I wish you an excellent reading and I hope you'll have as much fun reading this book as I had writing it!
>
> **— Pascal MARTIN**

[11] https://aws.amazon.com/developer/community/heroes/pascal-martin/
[12] https://leanpub.com/developper-une-extension-php
[13] https://php7avance.fr/
[14] https://twitter.com/pascal_martin
[15] mailto:contact@pascal-martin.fr
[16] https://blog.pascal-martin.fr

2. Our platform, our project

This chapter was written in the summer of 2018.

2.1 Our applications

2.2 6play: frontend, back and videos

Our 6play platform, which we internally call *"6play v4"*[1], is composed of a set of projects that work with each other.

On the *backend* side, we manage services, workers and applications, which communicate through API calls or message queues.
The services are called by the *front* applications or by other services. Our contributors operate two back-offices to publish videos or fill in their information. Workers process messages from queues.
On the *frontend* side, 6play.fr[2] is a Web application. Other *front* applications exist for Android and iOS or for IPTV Boxes[3].
Let's count about fifty projects actively maintained for applications and services developed by M6 Web. Or about twenty if we only count those we work on on a daily basis. Deploying them, hosting them and maintaining them in operational conditions is a big job!

We host tens of thousands of videos, which can be viewed from our applications in a dozen formats. Every month we serve more than ten petabytes *via* several CDNs.

Our traffic peaks are in the evenings between 8:30 and 11:00 pm and there is a smaller peak at lunchtime. The graph below shows the evolution of the *load-average* of one of our servers over 24 hours. The shape of these curves is the same on almost all our servers, for CPU consumption, the number of requests, or messages in queues:

[1] The v3 was "M6 Replay", which you may have heard of if you are French. Previous versions… Well, I don't have a lot of colleagues who remember them: it was a long time ago! Some of them were in Flash.

[2] http://6play.fr

[3] The "boxes" that ISPs provide: Freebox, Livebox…

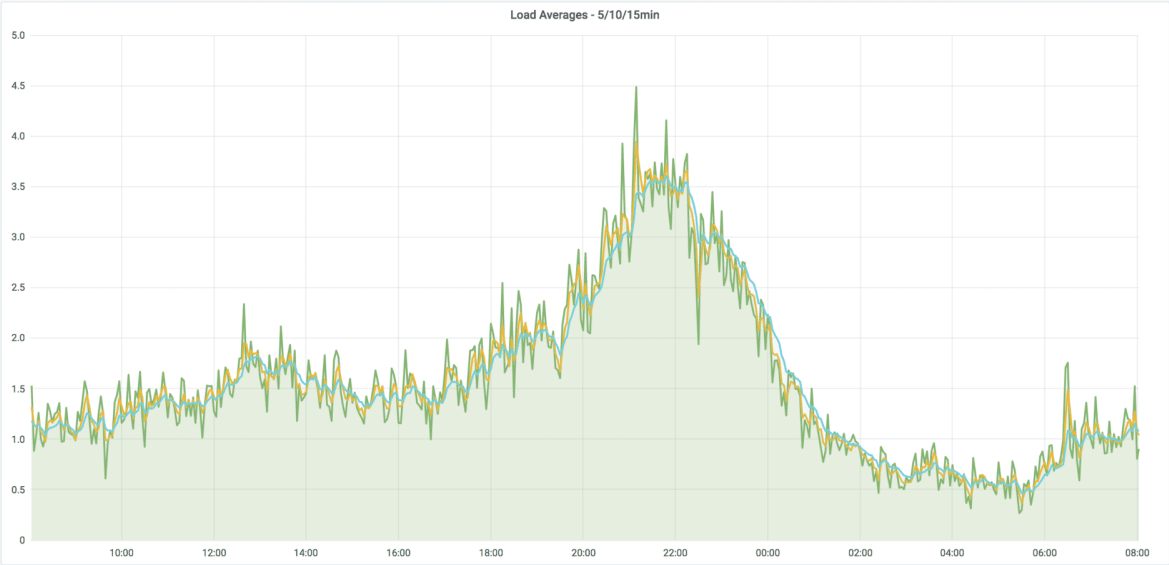

Figure 1 - Load average, the evening peak

This peak *at night* means we are free enough to work as we wish during the day: carefully, but going forward.
On the other hand, it also means our platform is most likely to suffer from potential problems - including performance problems - when no one is at the office.
So we need to do a good job and not deploy anything without thinking. This includes systematic *pull-requests* and code reviews, automated testing, continuous integration… Good practices, in place for several years, that we will not question in this book.

2.2.1 Our technical context

On the *backend* side, our applications are mainly written in PHP with the Symfony framework. We upgrade regularly, to benefit from the evolutions of these components and to not fall behind. During 2018, our new projects are developed in PHP 7.2 with Symfony 4.1. Our old projects are all in PHP 7.1 with Symfony 3.
On the *frontend* side, our 6play.fr Web application is developed in Javascript with the React framework. We use other dedicated technologies for mobile applications and IPTV boxes.

On both sides, we log a lot to our ELK stack and our developers use these logs on a daily basis.
We also benefit from a *Graphite*[4] stack which contains hundreds of thousands of metrics. They are used by *Seyren*[5] to trigger alerts in case of application problems. Some of them are visible on TV screens in our open-space: each team has customized its own dashboard to show in real time the health status of the projects it manages.

[4]Graphite stack: statsd, carbon, whisper, graphite and grafana toolkit.
[5]An open-source alerting tool that is no longer actively maintained. We're going to migrate our alerts to Prometheus, I'll talk about that in another chapter.

Figure 2 - The board of one of our teams

The sysadmins configure the servers with Ansible, with a few tens of thousands of lines of playbooks.

Application deployments are triggered by developers, whenever they want[6]. They do this through *GoLive*: a web application (developed internally) where they choose a project, a branch and a target environment (previews, preprod, prod…) to launch a deployment. *GoLive* triggers our *Deployer*: Capistrano recipes that know how to deploy all our projects to our *on-prem* infrastructure.
The graph below shows the number of production deployments triggered per day in June 2018[7]:

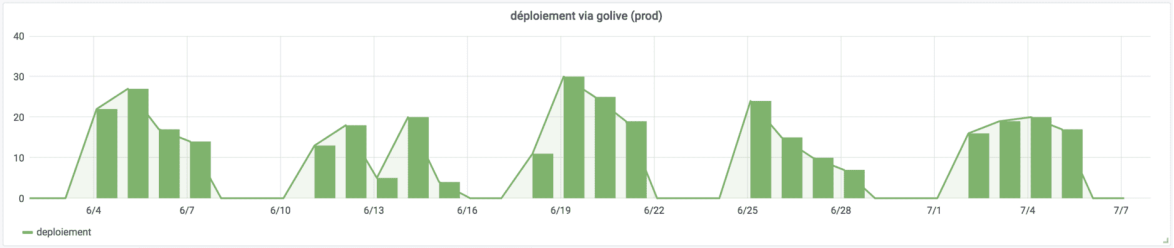

Figure 3 - Number of production deployments per day

We run several dozen deployments a day, all at the initiative of a developer, without ever waiting for *managerial validation* or going through a *process* of this type.

[6]Same thing for rollbacks, by the way: developers run their application deployments to production and don't need any external intervention (support, system administrators…) to carry them out - as long as they don't need to change the infrastructure.

[7]Outside of a holiday period, therefore.

We usually don't deploy on Fridays, to avoid unpleasant surprises at 6pm or during the weekend. It is a choice we made for *peace of mind*, even if it seems questionable.

Golive and *Deployer* work well and developers are quite attached to them, but they are no longer actively maintained[8] and don't know how to deploy to *The Cloud* (to Kubernetes). We no longer wish to invest in these old tools and, as part of our migration, we will replace them with a more modern solution - an open-source tool, ideally.

2.2.2 Organization and teams

In 2018, we were twenty-five *backend* developers, as many for the *frontend*, a team of five *ops* in Lyon with reinforcements in Paris, a *data* team of about ten people, a few scrum masters, a team dedicated to video and another one to quality of service, plus product owners…
The teams in each division (*frontend*, *backend*, *data*, *video*, *ops*…) are coordinated by a manager.
So there were about eighty-five of us on our Lyon open-space. Plus service providers and subcontractors who did't work on our premises or our many colleagues who worked on projects other than 6play!

The development teams, both *backend* and *frontend*, are made up of four to six people, including one lead-developer.
They follow agile methods, usually a variation of Scrum, with two-week sprints. The sprints of all the teams are synchronized, except for a few rare teams - including the *ops* - which have chosen to start their sprints staggered to better manage needs, dependencies and exchanges.

In this book, I will focus a lot on the *backend*, since I am myself in one of the *backend* teams and I have been a *backend* developer for a long time. But I will also talk, of course, about *frontend* and how we migrated them to *The Cloud*.

2.2.3 The International

In 2017, other RTL Group television companies in Europe, who did not wish to develop and maintain their own video platforms, chose to rely on M6 Web. Since the first quarter of 2018, our 6play platform also powers play.rtl.hr[9], www.rtlmost.hu[10] and http://www.rtlplay.be[11] - in Croatia, Hungary and Belgium respectively.

Our services and applications, designed for a single country, did not have a translation mechanism and only managed video from a single provider. We had to adapt them, sometimes extensively. We have:

- added a notion of *"customer"* : `6play` is a customer, just like `rtlbe`;

[8] No one here knows Ruby well anymore nor can write a Capistrano recipe…
[9] http://play.rtl.hr
[10] http://www.rtlmost.hu
[11] www.rtlplay.be

- internationalized the *front* where you watch our videos, as well as the contribution back-offices;
- and introduced *feature-flipping* on applications that need to be customized - the web front, in particular.

If you want more information, watch the video of the conference "De 6play.fr à une plate-forme internationale : retour d'expérience" (FR)[12] given by Guillaume Bouyge, lead-developer of the *backend* team created for this project, at Forum PHP 2018. You might also be interested in the video of the conference "Une donnée presque parfaite" (FR)[13] presented by Benoit Viguier at Forum PHP 2016.

On the other hand, we didn't change much in terms of hosting: we added a few servers or CPUs and RAM, since the load would increase with these three new countries. But that's it. These three new customers, from countries with smaller populations than France, brought traffic our infrastructure could handle.

2.3 An aging infrastructure

A few years ago, our *on-prem* infrastructure was pretty nice. But it dates from 2012-2015 and is therefore not quite up to date in 2018 anymore.

Most of our machines were purchased several years ago and have either reached the end of their warranty or their limit in terms of capacity[14]. Maintenance costs are increasing. We have voluntarily limited renewals for the past two years: we knew that our migration to *The Cloud* was approaching. Some older machines are still running on old Linux distributions[15]. We have upgraded some components (like PHP, Elasticsearch and Grafana) but not others (we are still running MySQL 5.5). Our monitoring and alerting stacks, although meeting our needs, are based on components that are less used today and are, for some (statsd, Seyren), not maintained anymore.

Here is the CPU consumption of our virtualization cluster, from August 10 to September 10, 2018:

[12] https://afup.org/talks/2776-de-6play-fr-a-une-plate-forme-internationale-retour-d-experience
[13] https://www.youtube.com/watch?v=kSlYXpezjIQ
[14] Especially for some network equipment, which showed weaknesses during our consultation peaks.
[15] Our newer machines are running Ubuntu 16.04, but we still have Ubuntu 14.04 and even some rare 12.04.

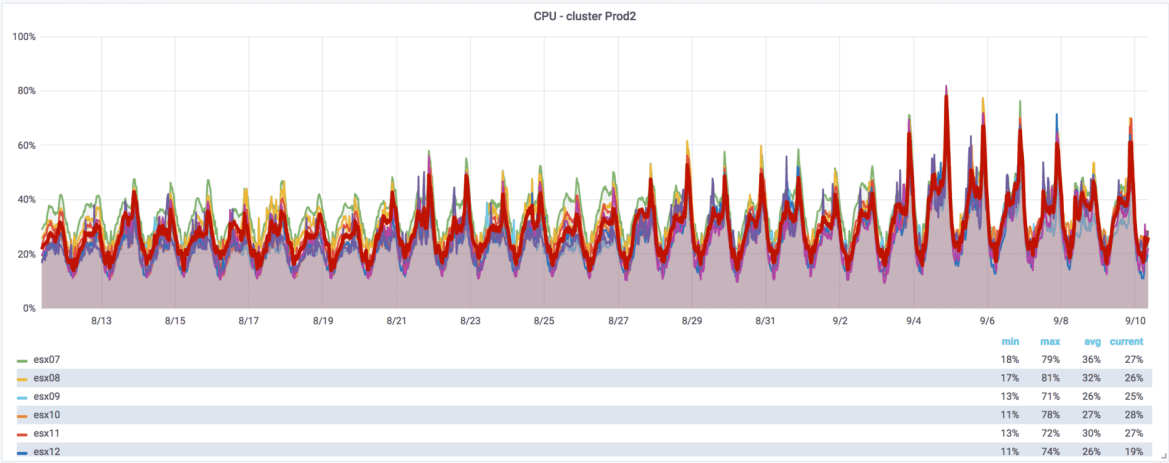

Figure 4 - CPU consumption of one of our virtualization clusters

During the summer holidays, our infrastructure was underused: even during our daily peak load, we never reached 60% CPU usage.
However, once the holidays were over, the load came back up and, some nights, we exceeded 80% of CPU consumption! This is a good sign: our platform is successful! But we won't be able to add a new client or a new application that is a bit heavy on this infrastructure.
Some very busy nights, some components of our virtualization cluster could even go up to more than 90% of CPU consumption!

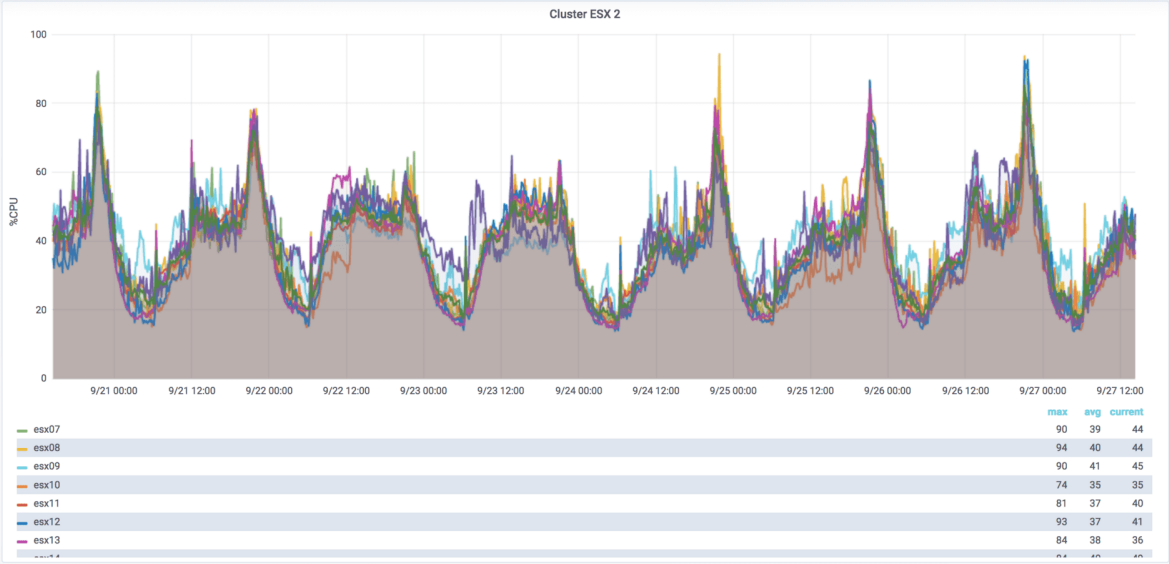

Figure 5 - More than 90% CPU on a virtualization cluster

Our very seasonal traffic means that working with constant capacity is not optimal. We could save a lot of money by changing it throughout the day and at different times of the year.

2.4 The future is coming

As developers or system administrators, we share a desire to improve and evolve. Note this is *evolution* and not *revolution*: it's not about *breaking everything* for fun!

Our vision is to be hosted almost entirely in a *public* cloud and to use *managed services* when possible.
We are willing to adapt our applications to use a managed service or features we didn't have *on-prem*, such as on-demand provisioning.
However, we want to keep what works well: agile processes, pull-requests, automated testing and continuous integration, deployment by each developer when they want, monitoring and alerting covering a huge surface area… And also the reliability that our customers are used to.

M6 Web and our 6play platform are not alone on this project: the RTL group is a shareholder of M6 and other channels and companies in Europe. It guides some of our choices and part of our work benefits our colleagues in other countries.

2.5 The DevOps team

Early 2017, M6 Web was considering the creation of a *"DevOps team"*. My hiring in October made this idea a reality: I was the first member of this team, before being joined by a second person in the spring of 2018, once the migration to *The Cloud* project was well launched.

First of all, since this question comes up regularly : *"why DevOps and not SRE?"*
The "SRE[16]" term was popularized by the book Site Reliability Engineering[17] co-authored by several Google engineers. But, at least in France, "DevOps" still has the advantage of being better known, especially beyond the technical teams - so using it facilitates exchanges.

Next, a question that is probably more important and that I asked myself at the beginning: *"why a DevOps team and not DevOps integrated into the development teams?"*
Several answers:

- A team means dedicated people (and therefore a budget). This highlights a strong will to *"do something new"* and associates additional resources to it.
- The members of this team are not assigned to a development team working on projects with a product-owner. They are therefore less likely to get *temporarily* caught up in these projects. All too often, companies promise to *improve their technical stack*, but since the interest is mostly long-term and not always well understood, these improvements are systematically postponed. With a dedicated team, this is less the case.
- Finally, the members of this team have a *"free electron[18]"* role and can more easily work with any other team.

[16] SRE: "Site Reliability Engineer."
[17] https://landing.google.com/sre/book.html
[18] When I met the CTO of M6 Web in a job interview, he used that term ("électron libre", in French) to describe the role I could play. In retrospect, that description was pretty good!

Creating a dedicated team may not be possible in your company and this approach does not entirely correspond to devops principles. But, in our current environment with a big project that involves both *devs* and *ops*, it works pretty well.

What are the roles and missions of this DevOps team?

- To help, train and accompany the teams, on our existing platform and for our target: *The Cloud*. Example: to deploy a project to Kubernetes, together with a developer from the team responsible for it.
- Test solutions, implement tools and best practices. For example, it worked on the deployment of the first project to Kubernetes and validated that what was implemented would also meet the needs of other projects (which will certainly re-use what the first one did).
- Pilot the deployment of new projects to *The Cloud* and the migration of others. This includes prioritizing what needs to be prioritized so that each project has its own hosting before its release date.

In short, the DevOps team exists to provide expertise and invest time. Its ultimate goal is to make itself sort of redundant[19], once the tools are in place and the development teams are trained.

2.6 A public cloud, Kubernetes

2.6.1 A public cloud

This book focuses on the migration of hosting of the 6play platform to *The Cloud*. To be sure of what we mean by this, I'll reproduce an excerpt from Wikipedia Cloud Computing[20] :

> Cloud Computing [...] is about harnessing the computing or storage power of remote computer servers over a network, usually the Internet. The servers are rented on demand, most often by usage bracket, according to technical criteria (power, bandwidth, etc.), but also on a fixed-price basis. Cloud computing is characterised by its great flexibility: depending on the level of competence of the client user, it is possible to manage the server himself or to be satisfied with using remote applications in SaaS mode.

There are three types of clouds[21]:

- *Public* cloud, available to the general public - including businesses.
- *Private* cloud, specific to an organization that will handle it itself.
- *Community* cloud, where the infrastructure comes from members sharing a common interest.

In our case, we are going with a *Public* Cloud. Some of the best known providers are *Amazon Web Services*, *Google Cloud Platform*, *Microsoft Azure* or *OVH Public Cloud*.

[19] "Being useless" on existing projects does not mean being useless to the company. On the contrary, once tasks have been automated or managed by other teams, the DevOps team can dedicate its time to other projects ;-)

[20] https://fr.wikipedia.org/wiki/Cloud_computing

[21] In Le Plan Copenhague, the French version of this book (you are reading its translation to English), I often say "cloud", but I also use sometimes the French term "nuage". In an English edition, this is somewhat lost, of course :-(

2.6.2 Managed Services

In addition, we have decided to make good use of the *managed services* provided by our *Public Cloud* provider.
Instead of renting machines on which we would install a database server (which we will have to maintain and administer), we prefer to rent a *database service* and entrust the responsibility for its management to our provider.

For a database such as MySQL or Postgresql, the proposed service is compatible with the open-source versions we were already using.
For other services, the solutions proposed by *Public Cloud* providers may be proprietary, specific to each one and incompatible with each other. If we develop software using one of these services, we will not be able to deploy it with another provider. We have taken this factor into account and have chosen to accept it: we are willing to partner with a provider if the service they offer is of high quality and relieves us of certain tasks.

For example, some of our applications used Apache Cassandra. We do not wish to administer Cassandra on rented servers in *The Cloud*. We are therefore willing to adapt these applications to take advantage of a managed service offered by our provider, if it meets our needs. The development load may be high, but it will be less than the time previously spent managing the service or servers.
Another example is a job queue. We use a lot of them, intensively. *On-prem*, we work with Redis or RabbitMQ. For *The Cloud*, we will adapt our applications to exploit our provider's job queuing mechanism. The development load is quite low and the gain is obvious: no sizing problems and no servers to manage.

We therefore aim to exploit the flexibility of *The Cloud* for services. We hope to gain in performance (on-demand or automatic provisioning) and availability (a developer who needs a service or wants to test another one clicks a button and *voilà*).
No more need to interrupt ops to create a database in emergency. Or no need to wait a few hours (or days in the worst case) for a ticket to be processed.

Of course, this flexibility comes with new issues, especially in terms of monitoring consumption and costs. If anyone can create servers, launch services or resize them, how do we know what is actually being used, what could be stopped, and where savings could be made? These are questions we will have to answer and matters we will talk about in a few chapters.

2.6.3 Infrastructure as Code

For our developers, everything is under Git, on Github[22]. No code is directly committed to `master`, development is done on a branch, which is reviewed by at least two people before being merged.
For our ops… Most of our *on-prem* infrastructure was managed via Ansible, and the principle of using commits and pull-requests was already well respected. But it was not always applicable, especially for network equipment configurations or firewall rules.

[22] We host an instance of Github Enterprise internally, which allows us to continue working even when Github is down - it happened more often a few years ago than it does now.

One of the objectives of our migration to *The Cloud* is to generalize the principle of *Infrastructure as Code* ("IaC"), which we already appreciate for certain components: all infrastructure elements must be described by readable and versioned files. As for our applications, any modification must be done through a branch and a pull-request reviewed by at least two people.
Our goal is a reproducible infrastructure: our development, staging and production environments will be created from the same files. Configuring the production environment becomes fast and risk-free: just reapply what has already been used in staging, adapting a few options (such as server sizing) in a variable file that has also been reviewed and versioned. We won't have to spend two days trying to figure out why a service is slow in production when it works very well in staging - just because someone changed an option on a server without ever documenting the change.

Of course, for our infrastructure to be 100% reproducible, we all have to respect this *Infrastructure as Code* principle. We have to be rigorous and none of us should change parameters *by hand* on a server or in an administration console!

A second objective is to give our developers control over their infrastructure. If they can create and configure servers or services themselves, they will be able to iterate faster than if they have to write tickets and wait for them to be processed. On this point, we will get a bit closer to the *DevOps principles*!

2.6.4 Kubernetes

Until now, our deployment chain has been running `git clone` and `composer install`, preparing the virtual hosts and flipping a symbolic link, while keeping the old version of the application to make it easier to rollback. It performed these operations directly on the production servers, handling them all in parallel. And it worked very well!

Having said that, exploiting the advantages of *The Cloud* means rethinking our deployment process. Indeed, if we want new machines to be launched automatically during a traffic or load peak, the approach of *deploying on machines that exist at the time of deployment* is no longer viable. Instead, we need a mechanism where new instances are launched by *fetching* the current version of the application.
A few years ago[23], we would probably have gone with a deployment chain that would have created disk images to push them to storage in *The Cloud*. The creation of instances would have been configured to boot from the disk image of the current version of the application.
Today, we have favored a container-based approach. Our deployment chain will create a Docker image and push it to a registry. The runtime environment will use the image of the current version of the application when it needs to create a container. We will use a container orchestrator to execute and manage them. The solution we have chosen is Kubernetes.

[23] Actually, we still have a production application that follows this principle - it was deployed in *The Cloud* before we chose Kubernetes. We'll get it to fit into the mold in a little while, so we won't have an application that isn't deployed like the others.

2.7 It was a bit fuzzy!

Reading the previous pages, you may have had the impression our project was well framed and defined? If so, it's certainly because I wrote all this almost a year later, when we had made good progress - and it's easier to look in the rear-view mirror than to predict the future!

In fact, in November 2017, our project was only defined in broad outline:

- We knew we wanted to migrate almost all of our infrastructure to *The Cloud*.
- We were seriously considering deploying containers, using a container orchestrator. Probably Kubernetes, but to be validated.
- We didn't want to lose quality on our applications, work methods or tools.

And, uh… And that's pretty much the starting definition of our migration project! Everything else, everything else I'll talk about in the rest of this book, was built up as we went along.
Sometimes we planned a few weeks in advance by preparing *stories*, detailing them and carefully listing the points of their *Definition of Done* before putting them in a sprint. Other times, we started working on *stories* that only read *"it must be possible to do X*[24]*, let's find out how to do*"! Some gave birth to others, more detailed and with a clear vision of how to do it.

To illustrate how we were throwing ideas on the table: I found two pages of notes taken during one of the first discussions around this migration topic. It was November 20, 2017, three weeks after I was hired at M6 Web. I had barely discovered our applications and I was still far from understanding the magnitude of the task that their migration would represent! In these notes, I found these points:

- *"Creating a build & deploy pipeline from code to containers."*
- *"Deployment on GKE with an already established spinnaker-like tool"* and *"canary / smoke / red/black"*
- *"Being able to roll back a deployment"*
- *"We have to go to production quickly […] to get some feedback (bandwith, stuff*[25]*, etc)"*
- *"Infra: graphing/monitoring/logging"*
- *"Building crash/test scenarios, how k8s behaves, monitoring, alerting etc"*
- *"APCu*[26]*, during a production deployment, the apcu cache is emptied, what does it look like in real life?"*
- *"Type of node to choose. Commit on CPU & RAM"*

From the very beginning of the project, the CI/CD chain appeared to be of prime importance: what we had *on-prem*, even if not quite modern anymore, worked very well and was used intensively. We were already thinking about monitoring, alerting and logging: here too, what we had *on-prem* did the

[24] With "do X" ranging from "writing a healthcheck of an application deployed on Kubernetes" to "thinking about how to migrate an application without downtime" to "collecting metrics about the health of a Kubernetes cluster and bringing them up to the on-prem Grafana we've been using for years".

[25] Yeah, I really wrote down "stuff" (Well, as those notes were in French, I actually wrote the French word "truc")…

[26] A cache mechanism often used in PHP, where the cache is stored in memory inside the PHP process - when the PHP process dies and a new one starts (like when you deploy a new container), the cache is lost.

job well, at least from a functional point of view. We wanted to iterate quickly, to go into production very fast, while at the same time taking the time to understand how Kubernetes behaved when there is a problem. We were aware that there would be real thinking to be carried out in order to optimize costs. But also, right in the middle of it all, we looked at a micro-detail of cache management. In short, we were far from the perfect, well-defined *roadmap*.
And on a project of this scale, that's normal!

> One year later: we still don't have a perfect solution for CI/CD (but we have an acceptable solution while waiting for something better), we chose (to move forward faster) to ignore the question of cost optimization this year and will come back to it next year, we have made good progress on monitoring and alerting but not yet on logging. And we have made progress on the management of Kubernetes' problems, as we have practiced - and we still have work to do!
>
> Read the rest of this book and you'll know more about how we've moved forward, bit by bit, on all these points, keeping our project in mind ;-)

2.8 In summary

We manage dozens of applications and services. They form the platform that powers the replay and VOD sites of several RTL Group channels - including M6 in France. In 2017, it was hosted *on-prem*, on an infrastructure that was beginning to show its age and limitations.
We want to migrate our existing applications and deploy the new ones to an infrastructure in a *Public Cloud*, managed according to the principle of *Infrastructure as Code*. We will use managed services and deploy our applications under Kubernetes, except when it makes no sense or when another solution is better suited.

3. Discovering The Cloud and Kubernetes

This chapter was written in the summer of 2018 and covers the period from November 2017 to May 2018.

3.1 Why "The Cloud"?

Why move to a *Public Cloud* when we were hosting everything *on-prem* before?

Infrastructure as a Service (or IaaS) is synonymous with flexibility: with a few clicks[1], we start up new servers or change the capacity of those we already have. This allows us to best respond to variations in the traffic on our applications, with a peak every night at 9pm or when we are broadcasting a football match or a popular show. We also experiment more freely with new services and iterate faster on new projects.
Hosting in *The Cloud* where we don't manage our servers will also mean, when we've done things right, less on-call and less intervention, including at night. If the infrastructure repairs itself thanks to disposable servers, it will be enough to (automatically) destroy a server that malfunctions for another one to (automatically) start and… problem solved! We will still have to investigate *"why"* the next day, in working hours and without the pressure due to *"production is down"*.

The Cloud is the direction the market is taking: servers are becoming a commodity and our job, as a company, is not to manage servers. We prefer to entrust our ops with tasks with higher added value than racking servers or replacing disks.

Migrating our infrastructure to *The Cloud* and considerably reducing our *on-prem* hosting causes ops to wonder: their job is changing, they have a lot to learn or relearn.
Asking developers to manage their infrastructure - or giving them the freedom to do so - also leads to questions, both for themselves (*"will we have to do this on top of that?"*) and for system administrators (*"will we still be needed?"* and *"will we be held responsible in case of an incident?"*).

> After a few months, the migration of existing projects has progressed well and new projects have been deployed directly to *The Cloud*. Things are going pretty well.
> Developers are moving faster and more freely, thanks to the more *DevOps* approach. They saw that they weren't alone to work on their infrastructure and that ops didn't just hand the baby over to them.
> Ops discovered new tools and a new approach. They made a big effort to relearn, which is bearing fruits: this new, more flexible infrastructure facilitates architectural evolutions and gives more room

[1] Or a few commands, since we won't be using the web management consoles.

for error.

3.2 GCP, AWS, ...

The vendor selection stage follows the decision to migrate. We targeted those offering the most functionalities as well as the best known ones. We considered that those whose services would be the most successful would probably also be those for which we would find the most support (open-source tools, community) or skills when recruiting.
Following an initial study and observation of practices within the RTL group, we focused on *Google Cloud Platform* ("GCP") and *Amazon Web Services* ("AWS"), with which we had already worked for ad hoc needs or overflow. We didn't focus on *Microsoft Azure* because we didn't have any "Microsoft" history in our platform, although from a distance, the service might have suited us.

To begin with, we met sales engineers who presented their solutions and tools. We in turn presented them our applications and needs. And we exchanged on the solutions that could be adapted for us. Then we made some POCs[2]: we tested the services that were supposed to meet our needs. We wrote programs working with these services, to validate their PHP clients[3] and to make sure they met our needs, performance included.
In particular, we tested three types of managed services corresponding to features we use intensively:

- The *"relational database"* service: RDS at AWS and Cloud SQL at GCP, for MySQL and PostgreSql.
- The *"job/message queue"* service: SQS at AWS and Pub/Sub at GCP.
- And the *"key/value database service"*: DynamoDB at AWS and Bigtable at GCP.

For the first two, no problems on either side; we didn't have that many worries.
For DynamoDB, we had bad memories following tests done a few years before, but that service has improved a lot since then and now seems to meet our needs, even if we noticed a point of attention related to configuration and scalability[4]. We had more difficulties with Bigtable: its PHP client was in beta and not quite mature[5]. To drive the nail in, the 51 results when looking for "Bigtable" on Stackoverflow (the majority actually talking about big tables under MySQL) did not reassure us about real usage of this service by other people sharing their experience. We managed to write our POC with Bigtable, but these points didn't give us much confidence.

[2] We did spend a couple of weeks there, with one person almost full time on these POCs and some other people working on them on an ad-hoc basis.

[3] Most of our backend applications and services are written in PHP. So it is important to us that this language is well supported by our vendor.

[4] We're going to have to think very carefully about the configuration of the RCU and WPU (Read/Write Capacity Units), which determine how many requests per second DynamoDB will accept to handle - and auto-scaling, while functional, is not instantaneous.

[5] Bigtable's PHP client was only auto-generated code from the protobuf, it was poorly documented and, to use it successfully, I ended up reading examples in nodejs and JAVA...

I have already talked about it above: yes, we agree to bind ourselves to a Cloud provider, using its proprietary services that are incompatible with those of its competitors. We will even invest time to adapt our applications; for example, migrating from Apache Cassandra to Google Bigtable or Amazon DynamoDB will require more than 200 days of work.

What if we want to change suppliers one day? These 200+ days will have to be reproduced, but will represent very little compared to all the work to be done: training of the teams, rewriting of a part of the infrastructure, the migration itself…

Our tests showed that we were able to do what we wanted, in terms of managed services, at AWS and GCP. Some services seemed more mature at AWS, especially when looking at PHP clients. Others did not yet exist at AWS or were more mature at GCP. Among the latter, the Kubernetes service: at the beginning of 2018, Google's GKE service was working very well, while Amazon's EKS service was not yet open to the public - we will come back to this later. Permissions management seems to be a pain on both sides. The network seems easier to configure at GCP. GCP has a much better multi-region abstraction than AWS, where the regions are really isolated.

We also evaluated network response times for Google (Belgium and Germany) and Amazon (Ireland, Germany and France). We measured them from France (where our *on-prem* hosting and the majority of our platform's users are located) and from Croatia and Hungary, two other countries where our platform is available. Here are the results of these measurements:

Table 1 - Response time from GCP and AWS

Source	Destination	Time
Croatia	M6 Web	35 ms
Croatia	GCP Belgium	27 ms
Croatia	GCP Germany	20 ms
Croatia	AWS Ireland	42 ms
Croatia	AWS France	30 ms
Croatia	AWS Germany	22 ms
Hungary	M6 Web	27 ms
Hungary	GCP Belgium	23 ms
Hungary	GCP Germany	16 ms
Hungary	AWS Ireland	45 ms
Hungary	AWS France	28 ms
Hungary	AWS Germany	20 ms
M6 Web	**GCP Belgium**	**5 ms**
M6 Web	GCP Germany	11 ms
M6 Web	AWS Ireland	18 ms
M6 Web	**AWS France**	**1.4 ms**
M6 Web	AWS Germany	11 ms

The very low response times from AWS France and GCP Belgium to our *on-prem* hosting (noted "M6 Web" in this table) are very interesting, since we will have many API calls between our two hosting until the end of our migration. Less than 5 milliseconds means we can make between one and three

API calls per page, from one data center to the other, without slowing down our applications too much.

A few additional arguments made the difference:

- *Amazon Web Services* is the leader in the field. *Google Cloud Platform* comes in third place, behind *Microsoft Azure*. It is easier to find advice or to recruit someone who already has experience with AWS. It's also harder to justify choosing one of its competitors - even if we would have done so if another solution had been much better.
- We and our colleagues in other countries have already deployed applications and services on AWS. We've also used GCP and others have used Microsoft Azure, but AWS is the only one we all have in common.

And a few arguments didn't carry much weight:

- The cost of one provider versus another: we didn't analyze the cost of each provider to the penny - we couldn't have done better than estimates anyway, which are necessarily wrong.
- The question of a European supplier or a US supplier.

GCP and AWS seemed to be able to meet our needs and neither was a *bad choice*. **We chose Amazon Web Services**. Or rather, we chose Amazon Web Services as our *primary* Public Cloud provider. We still have the option to use another provider for services where AWS would perform less well than a competitor.

3.3 Kubernetes

3.3.1 Containers for our applications

We started talking about it above: our *backend* applications and services are mainly developed in PHP. This language and its philosophy are perfectly suited for deployment to disposable servers - or containers with a short lifespan.
Indeed, PHP's approach is to treat each request independently of all others: there is no application server keeping a state from one request to another. On the contrary, at the end of processing a request, PHP frees up all the resources allocated to it. The PHP process itself is usually configured to die[6] after a few hundred or thousands requests.
In a traditional deployment, PHP is not even responsible for processing HTTP requests. A web server, such as Apache or Nginx, is almost systematically positioned in front of PHP.
You can see this as a disadvantage or as a strength. You have to manage a Web server in addition to the PHP runtime environment, but you can share it between several softwares developed with different technologies or replace it with an alternative that better suits your needs, without being blocked by a Web server that would be integrated into the language or your application.

[6]Historically, the idea was to force resource release in case of memory leaks - more often caused by PHP extensions than by the language itself. This is less useful today (the most common extensions have been fixed) but the habit has remained.

At M6 Web, we use the Nginx web server, and PHP is deployed in FPM mode - communication between the two is done in CGI through a Unix or network socket.

For our *frontend* applications, we also have nginx, this time setup in front of a nodejs application with the React framework.
Here too, the server components that generate the pages at first display do not keep a state from one to the other. Users can therefore be sent to any server and a new server can appear or an old one disappear, without any impact. Our *frontend* applications are therefore also suitable for deployment to disposable instances - to containers.

3.3.2 A Container Orchestrator

Containers and number of containers that vary according to traffic means a lot of work to manage them. To mention just a few of the questions that arise:

- How do I always have at least two containers running my application, even if one of them crashed?
- How do I launch more containers when the load on the application goes up? How do I delete containers when the load on the application decreases? How to detect when load is too high or when it drops enough?
- How to direct requests to the first containers? And to those created afterwards?
- How to ensure that requests from application A are distributed to containers from application B, without having to reconfigure A when a container for B is created or destroyed?

Having to manage all this? It doesn't sound like a dream. This is one of the reasons why many developers work with Docker on their machine but have continued to deploy in production *the old way*!

A container orchestrator is the answer to these problems.
At the end of 2017, the solution that seemed to stand out from the crowd was *Kubernetes*. It is an open-source container orchestrator, based on the experience Google has gained with its internal tool *Borg*. The community around Kubernetes is huge, the tool and its ecosystem are incredibly alive and several *Public Cloud* providers offer it as a managed service.
We started experimenting with Kubernetes at the end of 2017: we deployed a few of our applications to test environments, to verify this orchestrator answered the questions asked above. Our colleagues from other countries, who were already using other tools (like *Rancher* or *Fleet*), were also trying Kubernetes. In the spring of 2018, we all agreed and chose Kubernetes.

Kubernetes is not *"easy"*.

It only takes a few hours to deploy a first application to Kubernetes… But it will take much longer, maybe months, before you have a fully functional stack (automated deployment, logging, monitoring, alerting, scaling, error recovery, staging and production environments…) you will be comfortable with - even when *something goes wrong*: debugging with containers and Kubernetes, is not always exactly like we used to do *on-prem* on *bare-metal* or virtual machines!

Without ops who really understand *the system* and *the network* or without developers involved in the deployment process and who really know how the application runs in production… Well, good luck!

The purpose of this book is not to present all the features of Kubernetes. You will find many sources of documentation on the Internet, as well as many articles, books and conferences. To name only a few:

- L'orchestrateur de conteneurs Kubernetes : introduction (FR)[7]
- Kubernetes 101, by Stavros Korokithakis[8]
- Kubernetes Design and Development Explained[9]

Here is a very short summary of the primitives we will often use:

- A *cluster* groups together all the machines Kubernetes works with.
- A *node* is a machine, often virtual, which is part of the cluster. A *cluster* is composed of *master nodes* where Kubernetes internal functionalities run and *worker nodes* where your applications run.
- A *pod* is the smallest unit you handle with Kubernetes. Your application runs, as one or more *containers*, in a *pod*. To meet the load or to improve availability, several *replicas* of the *pod* can be created.
- A *HorizontalPodAutoscaler* or HPA dynamically controls the number of *replicas* of a *pod*, often based on their CPU consumption.
- A *Service* exposes a *pod* on the network - whether it is the cluster's internal network or the Internet.

3.4 Kubernetes on GCP and AWS?

We first experimented with the Kubernetes managed service of *Google Cloud Platform*: GKE - *Google Kubernetes Engine*[10]. We deployed a mini-"Hello World!" before moving on to something more interesting: our own applications.

[7] https://les-tilleuls.coop/fr/blog/article/l-orchestrateur-de-conteneurs-kubernetes-introduction
[8] https://www.stavros.io/posts/kubernetes-101/
[9] https://thenewstack.io/kubernetes-design-and-development-explained/
[10] GKE was originally called "Google Kontainer Engine", before it was certified to use the name "Kubernetes".

We started with a *stateless* application, which occasionally calls one of our internal APIs through a VPN between GCP and our *on-prem* infrastructure. With five milliseconds of delay between GCP in Belgium and our Paris data center, we confirmed the possibility of migrating this application without immediately migrating the API it depends on. We then deployed an application dependent on disk storage (deported for the occasion to AWS S3[11]) and a database. No problem there either.

In either case, we only wanted to experiment on test environments: even if the applications worked, we were not ready to go live in production!
We had no automated deployment, no logging or monitoring, no alerting that could detect failures, no disaster recovery plan, no *on-prem* failover…
And only one person knew what had been done; no ops had yet been in the loop. And, I forgot: billing was not connected, our CTO paid the bill every month with his own credit card and passed it on as an expense account.

We killed *pods*, they were restarted, we triggered load tests on a CPU consuming application, new *pods* were automatically launched and then destroyed when they were no longer needed, new VMs were automatically turned on when ours weren't enough to run all the *pods*, the YAML configuration files are reasonably adapted to pull-requests… The two or three months during which we (intermittently) experimented with GKE confirmed our choice of Kubernetes.

Amazon Web Services only announced its managed Kubernetes service at the beginning of 2018: EKS - *Amazon Elastic Container Service for Kubernetes*. It was only available in preview for several months, before being opened to the public in June 2018.

For Amazon, a *"Generally Available"* service may not be available in all AWS regions! It is open to the public, but it may be halfway around the world, 150ms or more from your data center!
For example, when EKS switched to GA in June 2018[12], it was only available in the US West (Oregon) and US East (N. Virginia) regions. Similarly, when *Aurora Serverless MySQL* was announced GA[13] in August 2018, it was only available in the US East (N. Virginia), US East (Ohio), US West (Oregon), Europe (Ireland), and Asia Pacific (Tokyo) regions.

When EKS was released, the service was not at the level of GKE: it did not exist in Europe and extremely important Kubernetes features - such as automatic scaling - were not available! But the work, still in progress, provided by AWS to develop this much-awaited service undoubtedly helped tip the balance towards AWS.

After we decided to trust AWS (the unavailability of EKS not being blocking), we had to find out how to create and manage a Kubernetes cluster without doing everything by hand! We chose the Kops tool, which we will talk about later.
In the future, we will probably re-test EKS - especially once the service is available in the AWS Paris region.

[11] Yes, from another vendor… The application already had a working S3 connector, but not for the equivalent GCP service.
[12] https://aws.amazon.com/about-aws/whats-new/2018/06/amazon-elastic-container-service-for-kubernetes-eks-now-ga/
[13] https://aws.amazon.com/blogs/aws/aurora-serverless-ga/

3.5 At what cost?

> *"How much will our hosting cost?"*

This is the most difficult question to answer when considering hosting in *The Cloud* and at AWS. Indeed, resources are charged depending on usage. And you pay for everything.

With traditional hosting, you know how much a server costs per month, you know how much your network connection costs if your provider bills you for it, you know how much you pay for the power supply, for the extended warranty of the machines and for their maintenance, and for the sysadmins' on-call duties. There is still a variable cost, which you generally know how to evaluate: the interventions in case of incidents. Some hosting providers make your life easier, and you only pay a lump sum that includes all this.

At AWS, you pay the servers according to the number of minutes they are on and the type of instance (which determines the CPU/RAM capacity of the machine) you have chosen. The price of each type of instance is defined by the region where you turn them on (services are more expensive in the Paris region than in the US regions). You pay for gigabytes of reserved disks. You pay for outgoing (but usually not incoming) network traffic except when it goes to another AWS service (which may itself charge for its outgoing traffic, potentially at a different rate) in the same region. You don't pay for traffic within a zone, but you pay for traffic between zones - brutal when you deploy your applications in all three zones of a region to increase their availability: good luck estimating the traffic between your services, especially when you work with micro-services that communicate in HTTP (or SQL, or talk to Redis) in all directions and have never had to worry about that *on-prem*. For some services, you are additionally charged by the number of queries - with different prices for write and read queries (and maybe even a different price for *consistent* and *eventually consistent* reads). For others, you pay for the number of requests per second you provision, whether you consume them or not. You also pay for the load balancers in front of your applications, as well as the CDN used as cache. We cloud go on for a while, but I don't want to scare you more than necessary. As you can see, estimating costs is hell and it's almost impossible to guess how much your hosting will trully cost. Oh, I realize that I didn't say you could reserve instances over one or three years to pay less, nor that the price of *spot* instances can vary according to demand!

However, we can't tackle a hosting migration without at least an idea of the maximum budget it will represent. Try to present a plan to your boss and tell him you have no idea what it will cost you and I doubt the reaction will be *friendly*.

How did we proceed, then? We started by counting how many virtual machines were hosting our *on-prem* projects and we counted their CPU and RAM consumption on our hypervisor during peak traffic. We also took into account in our calculations the physical machines operated for some projects. We then counted how many mid-range instances we would need at AWS to meet the CPU/RAM requirements of each application. The same was done for services billed by number of requests.

For example, one of our APIs was deployed on 8 VMs and was consuming 16 GHz of CPU and 32 GB of RAM during our daily peak. We considered that two `m5.xlarge` instances (4 vCPU and 16 GB

RAM each) should be enough to host this application. Two instances at 0.224€ per hour represents 322€ per month. Same calculation for a few dozen other applications.
This logic is of course very naive: we don't know exactly how much power the servers provided by AWS deliver, response times will probably vary and applications may spend more time (consuming resources) waiting.
At the same time, this simplistic logic masks an important point: our traffic varies enormously throughout the day. We have a big traffic peak in the evening and a small one at noon. Our servers don't do much the rest of the day. If, as we just did, we count the CPU and RAM load during the evening peak load and apply it over the entire day, we should have a good idea of the maximum price we will pay for servers and services. Then, we should consume fewer resources where possible, by shutting down servers or reducing resource reservations at night and during the day.

For these calculations, we have not included the use of reserved instances (which Amazon says can reduce the bill by up to 75%) or spot instances (up to 90% off, again according to Amazon). We have counted instances and services at full capacity all day long, without taking into account the possibilities of on demand sizing we would have, either at the level of AWS services or through Kubernetes.
Why? We know that we can reduce our costs by optimising our use of resources. But this is something we won't actively work on in the first year: migrating to *The Cloud* is a higher priority than reducing hosting costs on *The Cloud* - because we have to reduce our *on-prem* hosting, which costs *a lot*, and we can't do everything at once.
We also considered, when announcing this maximum budget, that we would be able to optimize afterwards, that it would help us to host new services and new applications from the second year onwards - without increasing costs.

We followed this logic to estimate the cost of AWS for our production. Then, we naively counted that *staging* (pre-production) would cost 40% of the production price, that *development* instances would cost 20% of the production price, and that the *sandbox*[14] environment would cost 10% of the production price.
We calculted the cost of non-optimized hosting in *The Cloud* would cost about 1.5 times[15] the price of our *on-prem* hosting - but this does not include the renewal of the machines, which became necessary.

I can't wait until next year to reduce these costs by optimizing our use! We'll talk more about this later in this book.

3.6 First Migration Plan

We[16] experimented with GCP and AWS for a few months, writing POCs to validate the choice of managed services we were interested in, and playing with Kubernetes to learn more about the

[14] In the sandbox environment, anyone can run any service to test something. But a script will regularly (every weekend) run and destroy everything.

[15] There is no magic: a server you manage yourself will often cost less than the same machine from an external host. But the host will manage many issues for you (maintenance, power supply, network, ...). Add to this the promise of flexibility of The Cloud, where you can boot or resize machines in a few clicks - that too, it pays off!

[16] One person almost full time.

container orchestrator we were going to use. We are now quite confident about the direction we are taking.
So it's time to start thinking about a plan! How are we going to migrate all our applications? How are we going to limit bast-radius in case of a failure? In what order will we move applications? Within what timeframe?

We have divided our work into five sets:

- **Step 0**. "Preparatory" work: choosing an IaC[17] tool, setting up the logging, monitoring and alerting solutions, as well as an automated deployment mechanism.
- **Step 1**. Our resource-consuming applications, those that will benefit most from the elasticity of the *The Cloud*: *user-facing* applications and the APIs serving them. Let's count about ten projects here.
- **Step 2**. Our advertising-related applications: they too consume resources - but we isolate them in a distinct set, since they are the ones that make money and finance our platform.
- **Step 3**. Non-user-facing back-offices and APIs: another ten or so projects, which need less elasticity - but once the others have been migrated, we will have no interest in keeping these *on-prem*.
- **Step 4**. And finally, everything related to IPTV boxes - we have less control over these applications, for which we work with external partners and ISPs.

We want to start by migrating simple applications that don't interface with 36 others, micro-services, utility applications. This will allow us to discover the problems that are bound to arise.
Then, when we become more confident, we will move on to the big, resource-intensive applications. We have dozens of servers running all day long in our data center, when a good part of them are only needed in the evening during our traffic peak. We will therefore take advantage of the elasticity of *The Cloud*, through AWS and Kubernetes' managed services.
We must adapt several applications. For example, to store files on S3 instead of local disks, to take advantage of DynamoDB instead of Cassandra, or to use SQS instead of RabbitMQ. These developments have an impact on the migration schedule: an application requiring 200 days of work will not be able to switch to AWS immediately, while an application requiring only 15 days will migrate earlier. For others, the switch to a managed service like RDS MySQL, instead of MySQL managed by us, will be transparent.

We have considered deploying some applications at AWS without adapting them - for example, deploying Cassandra instead of switching to DynamoDB. But we chose not to go down this path to take advantage of the benefits of managed services: this migration is the perfect opportunity. But we keep this in mind if there are planning constraints.

Our planning imposes a strong constraint: our hosting contract lasts one year. We therefore follow the progress of our migration carefully: a shift of one month would force us to pay for another full year!

[17] In the spring of 2018, we decided to run Infrastructure as Code - but we didn't know which tool we would choose yet. We planned to test Cloudformation and Terraform.

In collaboration with the backend team manager and the sysadmins manager (also responsible for hosting), we have established the following migration schedule for spring 2018. Each given date corresponds to the time when the migration must be completed and is preceded by several weeks or months of work depending on the applications:

- **Step 0**: spring 2018
- **Step 1**: Second half 2018.
  - Four simple applications: mid-June 2018
  - First resource-intensive applications, requiring little adaptation: October 2018
  - Other resource-intensive applications, requiring a lot of adaptation: mid-December 2018
- **Step 2**: End of January 2019.
- **Step 3**: Early spring 2019.
- **Step 4**: mid-spring 2019 (partly in parallel with step 3)
- **And finally**: physical interventions to unplug the servers and deploy them in a new smaller room: before the end of June 2019.

If we respect this migration plan, we will manage to free up the room that we currently occupy in our Paris data center to keep only a few servers in a smaller room starting in July 2019.

This migration plan is **extremely optimistic**: it corresponds to a vision where everything goes well the first time, where developers and sysadmins can dedicate as much time as necessary to this migration to *The Cloud*, and where no new project will interrupt or slow it down.
Why such an optimistic migration plan when IT projects have a reputation for always being late and new projects are likely to come in along the way? To have a way to detect whether we're going to overflow. If, at the end of June 2018, we haven't made as much progress as expected, we will know, a year in advance, that the *deadline* at the end of June 2019 (the anniversary date of our on-prem hosting contract) will not be met.
Given the high price of our *on-prem* accommodation, if we need to extend our contract for another year, it is important to know this in advance!

> At the end of June 2018, we made a progress report… It showed that we would not be able to keep this provisional schedule. Indeed, despite our efforts (and we hadn't stayed three months doing nothing), we hadn't completed the migration of any of the first four simple applications! At best, we were hoping to migrate one in July…

3.7 In summary

Our project is no longer just to deploy in *The Cloud*, but on AWS - *Amazon Web Services*. According to our tests, this provider meets our needs in terms of managed services and IaC and we have a solution to manage Kubernetes clusters while waiting for the availability of the EKS service in Paris. We have also agreed with our colleagues in other countries to use AWS as our main provider.

Appendices

Acknowledgements

This book would not exist without the help of several of my colleagues at M6 Web / M6 Distribution / Bedrock. Thank you to all of you for your patience with my questions, for the time you spent proof-reading dozens of pages of drafts, as well as for your good mood and the quality of your work everyday!

Huge thanks to the company itself and to our management, who push us to work on projects as exciting as this migration, give us the means to carry them out and encourage us to share our experience.

For this English translation, I would also like to give special thanks to Veliswa Boya, AWS Hero[18], and Sofia Lescano, developer at Bedrock[19], for their questions, comments, corrections, encouragement, and lots of feedback!

The photo used on the cover[20] of this book is by Nick Karvounis. Thanks to him!

Finally, thanks to you, readers: these tens of thousands of words would be much less interesting if you weren't here to read them! Thank you for the discussions we've had around the subjects I mention here, thank you for your attention, thank you for your feedback and suggestions!

Timeline

I joined M6 Group on 30 October 2017. We sent 1% of the traffic of our first application to *The Cloud* on July 16, 2018. Our two largest APIs were migrated in spring 2019. At the beginning of 2020, I knew that we would still have work to do around *The Cloud* for at least a year or two ;-).

You can find here the timeline of the main steps of our hosting change. I hope it will help you understand the magnitude of the task that awaits you if you embark on such a project!

- 19 March 2008: launch of *M6 Replay*.
- November 4, 2013: *M6 Replay* is renamed to *6play*.
- 2014 - 2015: the technical base of *6play v4* is put in place. It is in these years that we switch to the Symfony PHP framework, that we implement Graphite (monitoring), Seyren (alerting) and ELK (logging). The infrastructure is created on a set of virtual machines and a few physical servers, installed and configured with Ansible. We create *GoLive*, a graphical interface above *Capistrano*, which allows each developer to deploy any application[21] in staging and/or

[18] https://twitter.com/Vel12171
[19] https://twitter.com/SofLesc
[20] https://unsplash.com/photos/ciHYaPb8lUA
[21] Continuous deployment is a practice we've had in place for a very long time, starting in 2014/2015, long before we migrated to The Cloud. At the beginning of 2018, our developers were performing between 20 and 30 production deployments every day!

production environments at any time, in three clicks.

- Second half of 2014: With the *Rising Star* show and a huge expected peak load, we deployed some services in *The Cloud* for the first time. The experience is satisfactory, but as our *on-prem* hosting is still new and under-utilized, this experiment will be discontinued along with this show.
- December 1, 2015: release of the *"v4"* of *6play*.
- During 2017: we know that our *on-prem* hosting will not meet our growing needs forever (our platform will start hosting rtlplay.be[22], rtlmost.hu[23] and play.rtl.hr[24] in the first quarter of 2018) - and the market is clearly starting to move towards *The Cloud*. We're thinking about it too.
- **Monday**, **October 30**, **2017**: my first day at M6 Web \o/. I am the first member of our new DevOps team! To begin with, the topic that will keep me busy over the next few months is the migration of our platform to *The Cloud*.
- Friday, November 3, 2017: I'm typing *"Kubernetes"* in Google - I just heard the word for the first time. Yes, I was starting from far away[25]. I spend the weekend reading some doc. By Monday, I started to grasp what it is, what it does, and here I am, like a kid in a toy store ^^.
- November 2017 - April 2018: we test (features, performances, costs) the services of different *Cloud* providers, package applications as Docker images and deploy some projects under Kubernetes. Count about one and a half people for six months.
- March - April 2018: we are trying to set up a *perfect* CI/CD platform[26] to allow our developers to deploy an application to Kubernetes *in one click*. One person has been working on it for two months, but can't come up with something that meets all of our needs…
- Spring 2018: we draw up a first hypothesis of roadmap for our migration to *The Cloud*. In particular, we identify four projects that should be the first to migrate. If we manage to migrate them before July 2018, then we are likely to be able to reduce our *on-prem* hosting in 2019. If these four projects are not migrated by July 2018, then we will know that we will have to extend our *on-prem* hosting contract, of equal size, until 2020 (to have time to migrate the dozens of other projects).
- Early May 2018: a colleague and I attend *KubeCon 2018* in Copenhagen. We realized that among the speakers who were talking about their migration[27] to - or deployment in - *The Cloud*, none of them seemed to have a *perfect* tooling or *mastery*. While debriefing over a beer, we agreed that on Monday when we got back to the office, we would *brutally* migrate our first application: it would be the best way to find out what would go wrong and learn! In the end, we did it in a slightly more secure way, but I kept the name of the idea: *"Le Plan Copenhague"*, which I translated to *"The Copenhagen Initiative"* ;-)
- May - June 2018: our entire sysadmin team dives into AWS and Kubernetes. We choose Terraform to manage our infrastructure. We create our AWS accounts and our first Kubernetes staging and production clusters (`prod1`).

[22] http://rtlplay.be
[23] http://rtlmost.hu
[24] http://play.rtl.hr
[25] In my previous lives, I was a developer ;-)
[26] Auto-scalable, that deploys to Kubernetes or elsewhere, that allows rollbacks, that has a nice and intuitive user interface, that is easy to update, that everybody can configure as they want…
[27] I remember one conference where two Spotify engineers were saying that they had migrated their first API when only a sysadmin could deploy to Kubernetes, from their laptop…

- Beginning of July 2018: we prepare HAProxy to perform a gradual switchover to *The Cloud* of our first application. In parallel, we write its Helm / Kubernetes manifests. The application is deployed (not publicly accessible) from our laptops, with the commands `helm` and `kubectl`.
- **July 16, 2018** : we switch 1% of the traffic of our first application to *The Cloud* . This is the application I mentioned in the chapter "a first migration". We're going up to 25% and then 50% in the following days.
- Second half of July 2018: we migrated - and only partially - only one of the four projects we had listed as *"to be migrated before July 2018"*. So we know[28] that we need to extend our *on-prem* hosting, at equal size, until 2020.
- Second half of July 2018: implementation of a basic CI/CD chain to allow our teams (including developers) to redeploy / update the project deployed in *The Cloud*.
- July - August 2018 : Learning and correction of the first real problems encountered in production. To learn more, read "the beginning of the problems".
- August 2018: I begin to draw up the plan of this book, then write the first sections. I talk about it to colleagues who agree: the idea of telling our migration is interesting.
- September - November 2018: implementation of logging and monitoring and alerting tools. Writing of documentation, first updates of Kubernetes in production, creation of a second cluster "`prod2`" (and migration, without downtime, of applications from the first to the second). You will find a lot of information and details in the chapter "a first stabilization phase".
- October 2018: deployment in *The Cloud* of the first project that has never been to our *on-prem* hosting.
- October 2018: we start using EC2 "spot" instances in our Kubernetes staging cluster, to reduce costs (~70% savings).
- December 2018: Migration of our image thumbnail generation service - I mentioned it here.
- December 2018: Developers have (read) access to the Kubernetes staging cluster.
- **December 18, 2018**: publication of the first six chapters of this book (French edition).
- February 14, 2019: we change CDN in front of our thumbnail generation service, causing a suddenly empty cache. We go from 6 to 282 CPUs consumed in 16 minutes. x47 in 16 minutes! *On-prem*, emptying the cache of this application was simply impossible!
- March - June 2019: Gradual switch of our catalog API. This is our first big API to migrate to *The Cloud*, we took our time and put this migration on hold for a few weeks to focus on other issues (see below). I mentioned it there.
- March 2019: using CoreDNS instead of KubeDNS, to reduce the DNS flickering we suffered from for several weeks.
- Beginning of April 2019: we order EC2 "Reserved Instances" (~30% savings) for the three *master nodes* of our production cluster + *six worker* nodes.
- Mid-April 2019: we switch our user preference API (at night, 4h service interruption) to production. Switch from Cassandra to DynamoDB for data storage. I talked about it this way.
- April 2019: Developers are now allowed to manage their infrastructure (AWS managed services) in our staging environment.

[28] This was to be identified early enough (hence the idea of identifying the first four projects "to be migrated before July"), to be taken into account in budgeting.

- June 2019: kube-aws/kube-spot-termination-notice-handler[29] is installed, in staging (and in production the following month), to properly manage *reclaims* of EC2 spot instances.
- Summer 2019: in about two months we go from *"we only use EC2 on-demand"* instances to *"we almost exclusively use spot instances"* for the *worker nodes* of our production Kubernetes cluster. This way, we reduce its cost significantly.
- July 2019: setup of an Ingress (HAProxy) in our Kubernetes cluster, instead of one ELB per application.
- July 2019: we start using AWS CodeBuild for build/testing/deployment of some applications, to relieve Jenkins (it was really over-loaded).
- End of August 2019: creation of the cluster "`prod3`" and migration of all applications from "`prod2`", without any downtime.
- September 2019: isolation of "critical components" (Ingress, cluster-autoscaler, Prometheus…) from our clusters to non-spot *worker nodes*, to prevent them from flickering too much.
- September 2019: installation and use of Loki for logs written on stdout/stderr of applications, instead of Cloudwatch.
- Beginning of October 2019: creation of the "`prod5`" cluster, with VPC-CNI as its network layer.
- Mid-October 2019: Use of many types of EC2 instances, to reduce the frequency and the number of reclaims of *spot* instances.
- November 2019: We install VictoriaMetrics for long-term storage of Prometheus metrics.
- January - February 2020: we are gradually switching the Web Fronts of our various customers. I discussed this in this chapter.
- Early February 2020: Implementation of our first AWS Config rules, to detect poorly configured AWS resources. Example: DynamoDB table without PITR, or without auto-scaling.
- Mid-February 2020: migration of our main backoffice. A few words about it here.
- *One beautiful day*: setup of a new *on-prem* (smaller) server room for the few services we don't want to migrate to *The Cloud*. And the end of our old, bigger, historical server room. On paper, our migration to *The Cloud* is complete.

In short, our project to migrate to *The Cloud* has been a long term one and not quite *"we migrated in two hours"*. I may complete this timeline in the future, as the last chapters of this book are written.

[29] https://github.com/kube-aws/kube-spot-termination-notice-handler

And now? Get the book!

You have read the introduction of my book *The Copenhagen Initiative*, two whole chapters and you've had the chance to see the timeline of our migration to *The Cloud*.

If these few dozen pages have - and I hope so - aroused your curiosity and if you want to learn more, you can buy this book in full version at the following address: leanpub.com/tci[30]

It is available in PDF, EPUB and MOBI electronic formats. New chapters and updates that may bring some corrections and additions here and there are of course downloadable at no extra cost.

> Once again, I wish you an excellent reading and I hope you'll have as much fun reading this book as I had writing it!
>
> **— Pascal MARTIN**

[30] https://leanpub.com/tci