The Tao of DevOps

A journey from programming languages to modern IT infrastructure to be a great professional

v0.1.0

Marcelo Pinheiro

Contents

Introduction

Let's dive into the journey

Hi. Probably you are considering to start a career as a DevOps Engineer or maybe already have experience in this position. In the modern era we are living now, with a plethora of options and abundant information on the Internet about this topic, maybe you will face some difficulties about how to start from scratch or developing all skills required to perform with excellence, gluing all knowledge and having a holistic view of all you need to learn and improve.

With the creation and implementation of the concept called *Ubiquitous Computing* by mobile devices (tables, smartphones, IoT devices and so on) most of the Computer Science fundamental knowledge has been abstracted in frameworks of all types for mobile applications to frontend / backend systems and even Operational Systems. Despite we don't need to worry about low-level implementations like handling C pointers and stuff like that in old times, it is causing a side effect: the foundation of Computation and Programming basic skills are being affected by the abstractions these frameworks and tools does for the programmer. The abstraction level has reached a limit which we even don't know what the operational system file system runs in some of these devices, all we need to do is simply have experience with a framework to do the job *using libraries and techniques on the scope of the tool we use*. That said, you need to develop your **Programming Skills** in some order to not be hostage to the tools you use.

In the first glance maybe you think I'm against modern frameworks, tools and libraries we use today. On the contrary, I am an evangelizer of most of all this apparatus; for example I use frameworks like Ruby on Rails for my personal projects, Terraform to handle infrastructure and others tools. Since 2000, when I started writing code, I can see how useful and battlefield proven these tools are to save time. In the LAMP golden era, I knew how manual was the effort to set up a Linux instance to run this stack. You needed to use hosting solutions or maybe configure an on-premise server or even a Virtual Machine to run your WordPress webpage. With the advance of Configuration Management tools, we found a gold pot to finally setup servers in an organized, secure and predictable way. If you write a good code using the combination of Terraform, Packer and Ansible you can bootstrap your LAMP server in a few minutes in the cloud, if you do not decide to use a 100% managed service. But the **Automation** is just one of the required skills you must master.

OK, just configuring a LAMP server is not enough. You need to develop one of the most underrated but critical skills called **Troubleshooting**. Let's suppose your MySQL database on your LAMP stack suddenly started to loss performance for an unknown reason; how you will fix it? There are questions we must ensure if is some SQL query that are degrading the server, out of memory, network bottlenecks, file system failure (yes, it occurs) or even a bug in your database server (personally I faced issues like that). How you can detect every problem I described? You need to develop your operational system

skill set to see the current status of the CPU/RAM, file system, application / database / OS logs, network, debug applications and so on. To master troubleshooting, you need to take the hand of a reasonable checklist and proceedings to help you in at least a little bit when something goes wrong at Saturday late evening, and you are on-call.

Well, nobody likes to wake up with smartphones screaming like police sirens if your application goes down. You can predict strange behaviors in your systems using **Monitoring** tools and techniques to being warned in a good time window to prevent an incident. But monitoring is not about only metrics, you need to create a holistic approach to generate meaningful metrics. You can (or better, must) instrumentalize your application to send both business metrics - let's suppose a number of successful subscriptions in your webpage - and behavioral metrics like the amount of CPU and RAM used, network latency for specific API calls and so on.

The great value of these metrics gives you the opportunity to go one step beyond called **Observability**. The DORA (DevOps Research and Assessment) research describes it as a tooling / technical solution to proactively debug and monitor your system. It helps you to measure key business metrics in order to make decisions - i.e a launched feature is being used or not - and overall system metrics to detect outages, unauthorized activities and so on. Here enters some of the most famous keywords like SLA, TTR and others we will cover in this book.

OK, it sounds good, but we didn't finish yet the required skills to be a great DevOps Engineer. Let's dive deep back into the programming skills to approach the most important skill you must have in your Swiss knife: **Systems Architecture**. Again, I will share my experience in this field to give you an example. When I worked as a Java programmer in my hometown a long time ago, I felt that bad gut feeling that everything you need to solve you must use Java libraries or tools written in this language. OK, we know the immense value of the Java Virtual Machine aka JVM, but the world is not moved by stateful applications, thread pools and database tables being used as queues or some sort of messaging. In the rise of Distributed Systems as we know today, we notice that we can compute things in asynchronous way without losing performance, or better: we can improve a lot. It may sound strange, but web sockets gave us a totally new approach to develop resilient, high available systems. Of course, several communities like Ruby, Python and others contributed with a ton of tools to reach asynchronous processing, from Apache Kafka message broker to Sidekiq, the famous Ruby asynchronous job scheduler. The born of NoSQL databases like MongoDB, Redis, GraphQL and others gave us the wisdom to use the right tool for the feature you want to develop or that annoying bug you need to fix. But, in the other side, using these tools just for the sake of hype can literally destroy your application or even worse, your business. You need to create a mind system to proper elect the libraries, tools, databases and other systems from the early beginning of your project to get a good balance in order to deliver an MVP or improve a legacy ecosystem.

Another crucial skill you must master is **Software Distribution**. This topic is immense, and we will

cover it as an entire chapter in this book, but to give you a little taste I will explain how hard was this in my times of C# and Ruby programming. For the famous Windows programming language, you needed to make a build in your local machine to generate a package (commonly a zip file with a few DLL's) to be deployable in the famous Windows IIS Server. In the other side (the sysadmin one), we had a teammate responsible to unzip these DLL's in some folder, copy them to the IIS folder structure, restart the IIS application and literally cross fingers to not break anything. With Ruby, I faced several deployment issues related with the famous *it works on my machine* provided by libraries installed in the developer's computer but not in the production server, causing a lot of rollbacks. To avoid the problematic rollbacks, I wrote a Ruby library called *gordon* to package our Ruby apps to be Debian compatible using .deb files (this library can generate RPM packages as well), based on a set of curated sysadmin conventions in a way you can simply install the application using `sudo apt-get install app=1.2.3`, execute a `sudo service app restart` and voilà, our app is up and running. Today, you don't need to do all the work of maintaining an APT or RPM repository server and package apps; Docker has revolutionized the way we distribute software today. All you need is a Docker base image, install a few packages (oh, look here the APT / RPM repositories still delivering software releases today), make additional steps in your Dockerfile and push the image to Docker Hub or a private Docker repository in a cloud provider. That said, if you want to master this skill, you must have experience with the traditional repo-based package distribution and how to *Dockerize* your applications.

To complete this journey, you want to deliver software as fast you can get. In a world of instant gratification, you must be agile enough to deliver a product from scratch and incrementally improve it based on the needs of your audience if you want to create a digital product or a new system to replace some legacy stuff. Here enters the holy **Continuous Integration / Continuous Delivery** best practices. This topic is huge and will be covered in this book, so I will be short here. In the old days of programming, how we ensure if a feature is working? By starting up a local copy of the application source code by using an IDE or maybe a Shell script, navigate into the feature we want to deliver or the bug we want to fix, finish it and take a look on the database to ensure if our updated code works. Of course, this was a very manual process that tends to be flawless at any moment. It was common forgetting to deliver a SQL script to create some additional tables when we tried to deploy into production, causing a lot of frustration and chaos when your deployment doesn't have any possibility to roll back to the previous state. Supposing our deployment was successfully made into production in terms of installation / setup, how we can prove everything is OK? Navigating into the feature / bug fix flow, making some CRUD operations… manually. With CI / CD, we can automate all this boring or critical business / system requirements in a pipeline that at least ensure your software is proper covered by Unit Tests, Acceptance Tests and - depending on how mature your software cycle is - you can even automate the deployment in an on-premise server, a cloud VM or a Serverless platform. To reach the excellence being a DevOps Engineer, having a solid knowledge and experience with CI / CD is mandatory.

Probably you be surprised how long this journey can be, and it is in fact. I must tell you, based on my DevOps Engineer career, it's one of the toughest professions I ever saw in my life. You need to be in touch what is happening now in both Development and Operations side, following the latest features provided by Cloud providers, new languages like Rust and V, new NoSQL databases popping up; you must be well-informed about the actual stack the market adopted. Take a look at Kubernetes for example; I saw the evolution of this container orchestration tool since 2016 until he won the battle against Docker Swarm, rkt and others. From AWS EC2 instances in Auto Scaling Groups until the ECS Fargate, a 100% managed container orchestration solution. That's a ton of content to develop and master. Being a DevOps Engineer is raising the bar up all the time to keep your experience sharp enough to attend the market requirements.

But don't worry how so long is this journey. Instead of exploding your head, enjoy the knowledge path. It is worth the effort, if you love to write code and deliver things with high quality and value. I guarantee that one of the most satisfying joys of life is writing an API from scratch and orchestrate everything I highlighted here in a **Continuous Deployment** pipeline. Come with me.

> *"I maintain that truth is a pathless land, and you cannot approach it by any path whatsoever, by any religion, by any sect."*
>
> *Jiddu Khrisnamurti*

Programming Skills

Know what you must master as a programmer to think beyond

Before digging into what really matter, I want to share my professional experience as a programmer to give you some insights.

I never had a computer in my childhood or adolescence. My first real contact with computers was in the high school, when we made some classwork that required to make some research on the internet. My first job was a graphic designer, using Adobe Photoshop and CorelDRAW and sometimes Adobe Page Maker; notice I didn't have any contact with programming languages before. After looking at my hometown job opportunities newspaper, I applied for a job to work with Macromedia Flash as a Web Designer. After getting the job, things became interesting.

It was 2000, in the emerging boom of e-commerce websites and hot sites using Macromedia Flash. Because we had only dial-up connections using modems with a peak of 128kbps, generating small SWF's (the Flash output file) was critical to deliver a good user experience for our customers. I notice some animations can be programmatically created instead of using Macromedia Flash object animations on the IDE, using ActionScript - an embedded script programming language very similar to JavaScript to

compute things. On that time, I started to learn how to write ActionScript code to decrease the size of the SWF's file size by applying some techniques that evolved *algebra* and some *physics* **functions**. Fortunately, I generated SWF file's sometimes 50% or even 60% smaller than the original one by using ActionScript programming instead of generating layers of animations, so the effort make a huge worth.

OK, sounds good but what the hell this has any connection with programming skills? Well, now I must show you one of the most famous and simple mathematics function of all time: the **Fibonacci Sequence**.

```
F(0) = 0, F(1) = 1
and
F(n) = F(n - 1) + F(n - 2)
for n > 1
```

If you never had contact with Fibonacci Sequence, I strongly recommend you to have a minimal knowledge of high school foundational mathematics to proceed as a programmer. It's a **must-have** skill you need to develop. Unfortunately, I notice that in the last decades the quality of high schools and even universities was dramatically decreased in a way that we need to revisit the fundamental mathematics with recent college graduates to mentoring them in programming languages. I'm not talking about Calculus, Physics and Statistics here; of course if you develop these skills it will be great to solve problems in code using algorithms and formulas from these fields, but before that you really need to understand the mechanisms of mathematical functions. The good news is: this is not so complicated.

That said, let's dissect the Fibonacci Sequence. If you pass zero, it will return zero. If you pass one, it will return one. For two and bigger numbers, you will see what we commonly say in the Computer Programming universe as *recursive function calls*. We will brief it later, but the greatest tip I must tell you is: **notice the patterns**.

The art of programming and using terminal commands are in fact:

> **You provide an input, the computer process it and gives you an output that you want.**

It may sound weird, but the cornerstone to be a good programmer is to **mastering the use of functions**. Whatever programming language paradigm you want or need to use, it's all about calling functions that will give you an output based on your needs. Let's dive into this Java code snippet of the famous **Hello World** below:

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Abstract `class HelloWorld` for a while, we will discuss it later in this chapter. Take attention to the following line: `public static void main(String[] args)`. For Java, you need to declare the method in order to execute a program. OK, in this code you can pass an array of String's to process it in your Java application, but pay attention to the pattern. I will give you another code snippet to train your brain to learn patterns; let's look at the Fibonacci Sequence written in **Elixir**:

```
defmodule FibonacciGenerator do
    defp comp_fib(0), do: [0 | 0]
    defp comp_fib(1), do: [1 | 0]

    defp comp_fib(n) do
        [h | t] = comp_fib(n-1)
        [h+t | h]
    end

    def calculate(n), do: hd(comp_fib(n))
end
```

Again, abstract what `defp` is in practice. But what you will have in common? **A function call.** - in this case `calculate` and `comp_fib`. In Elixir, we have a great feature called *Pattern Matching*, that I will cover briefly later - this language uses the Functional Programming Paradigm. But notice that **all code snippets you see here are basically function calls** programmatically speaking. Do you want another example? Let's dive into the arcane programming language **Turbo Pascal** with the same Fibonacci Sequence function implemented:

```
program fibonacci;

function fib(n: integer): integer;
begin
    if (n < 2) then
        fib := n
    else
        fib := fib(n-1) + fib(n-2);
end;

var
    i:integer;

begin
    for i := 0 to 16 do
        write(fib(i), ', ');
    writeln('...');
end.
```

Abstract the noisy Turbo Pascal syntax and take a look at `function fib(n: integer): integer` line. Again, *another function call* with a minor changes compared with the Elixir code snippet. Turbo Pascal is a Procedural Programming Language, what implies you to explicit declare

your code in a structured order to compile it and generate a binary; this is why it looks weird compared with the Java Hello World code snippet.

Probably you are asking yourself why I'm mixing a Hello World and Fibonacci Sequence algorithms in different languages to explain how programming works. There's a reason behind it and I will explain now, so reflect and internalize what I will say:

> **Computer Programming is a set of skills that blends mathematics, logic, linguistics and cognitive intelligence.**

Why Linguistics and Cognitive Intelligence are so important for programming

About Linguistics

My first programming language was ActionScript, when I did the optimizations I told you a few paragraphs ago. In my career, I got contact with these languages in the exact order since 2000:

- Macromedia ActionScript
- ASP (Microsoft Active Server Pages)
- PHP
- Macromedia ColdFusion
- Turbo Pascal
- Visual Basic 6
- Assembly x86
- C#
- Java
- C
- C++
- Prolog
- Common Lisp
- Python
- Ruby
- Golang
- Erlang
- Clojure
- Rust
- Elixir

- Lua
- Haskell

It sounds madness, but I wrote code using these languages since I started working as a programmer (and I did not quote databases and other tools). Despite my graduation gives me some advice about the nature of some languages quoted above, knowledge about the fundamental best practices writing code, developing / fixing real code teaches me a lot in an imaginable order of magnitude compared with my old days at the university. Being exposed to real software is one of the best tips I give to you, whatever language it is. You must master the ability of winning challenges by stepping out of your comfort zone and fixing a critical bug in an unknown language. Of course, this is a hard to swallow pill, but it's a liberating one.

Approach programming languages as an idiom you want to learn. In fact, despite I tried to find some researches in this field with no success, I notice the best programmers I got the opportunity to work with were polyglots in idioms and/or programming languages. They are more prolific in a programming language of choice, but they were capable of solving complex problems / develop features using not only one, but two or more programming languages if needed. If you stop to think, today a modern programmer must know what TypeScript is for the frontend and Node.js for the backend one. Despite both uses JavaScript behind the scenes, they are completely different in syntax. OK, you can see a mix of pure JavaScript inside TypeScript code because this is a super-set language, but in practice we are dealing with two different idioms. If you want a simpler example, consider Ruby on Rails in the frontend consuming Golang backends, it's very common to see today this kind of language's blending when launching a digital product. You must be prepared to work with this plethora of frontend / backend technologies and be adaptive, making right decisions in your architecture to scale up your system and measuring what is the best to the problem you need to solve wisely.

That said, if you want to develop your linguistics skills, be exposed to different idioms. If you speak English, start learning Dutch and German later. You will see some patterns along these languages with time, what will teach you in all sides. If possible, challenge yourself with a non-western language like Ukrainian, Czech or Serbian. For Cyrillic derivate languages, you need to transliterate every word in the character symbol's table to get the word in slavonic-like idiom, and now you need to translate it. It's a double-step proceed to learn the language and I confess that it will burn up your brain in the first moment, but as everything in life you must practice being prolific with. I strongly suggest you to learn a new idiom to improve your programming skills.

Probably you will ask why I'm suggesting learn a new idiom. It's simple: treat every language as an idiom. There are idioms that some words are easy, others that the word is complicated a lot - specially when dealing with German. But every idiom has their way to express words, and programming languages are just a way to express solutions by writing code based on the input -> output flow I mentioned before. Some languages, depending on how mature / robust their std (Standard) libraries are, will give you an

easy path to write an algorithm; another must be more verbose because **the design they approached requires it to be is**. That's why you see some implementations of code that are very short in some languages compared with all the Java verbosity you must create to deliver the same solution.

About Cognitive Intelligence

Cognition, by definition, is the ability to get knowledge based on the conditions in the environment you belong. Cognitive Ability is the aptitude everyone haves to individually interpret and take decisions based on their overall knowledge developed by your cognition capacity. Cognitive Intelligence is, in a short description, a tool set of thinking ability, abstraction, reasoning, memory, language, creativity and problem-solving.

Most of the challenges in Computer Science that tests you are basically composed by algorithms to be used, the overall architecture your systems will run and your ability to convert business features into source code. I'm not talking about building complex real-time software like rockets, avionics and military defense weapon software for missiles, but the mundane daily work to be made. If you want to launch a digital product, your Cognitive Intelligence will be put to the test in all these fronts:

- Algorithms
- Software Architecture
- Infrastructure
- Business Requirements
- Minimum Viable Product (aka MVP)

That said, you must blend all these knowledge fields all the time to write source code. If you have time enough to study better on this field, I recommend taking a look at the legacy of Jean Piaget in the Theory of Cognitive Development he wrote. He not only created an entire science field called Genetic Epistemology, but also inspired Seymour Papert by his thinking approach to develop an educational programming language called Logo. I strongly recommend you to increase your Cognitive Intelligence having contact with all fronts I mentioned above in a daily basis, developing one by one during your career as a DevOps Engineer. It's part of the job to wear a lot of hats in these knowledge topics but, taking one step a time to avoid burnout, you will create a holistic way of thinking that you will be grateful of and will speed up your productivity.

Now it's time to hands on start practicing.