# PATH TO SENIOR DEVELOPER

## Mastering Code, Leadership & Impact

Matteo Baccan

# Preface

The material in this book stems from hands-on experience, compiling a series of articles that I have carefully selected and developed over several years of professional work in the programming world.

These articles, originally published in Codemotion Magazine, one of the most authoritative technical information platforms for developers in Italy, serve as a testament to my professional journey and the insights I've gained over time.

The goal is to share reflections, strategies, and technical insights that could prove valuable for senior programmers—professionals who have already built significant experience but are always seeking new avenues for growth and improvement.

Each article represents a piece of an ongoing learning journey, born from challenges faced, mistakes made, and lessons learned during complex projects and challenging work environments.

Through these pages, I aim to offer not just technical knowledge, but more importantly, a broader perspective on the craft of programming. I share methodological approaches, best practices, and reflections that go beyond mere code.

I hope these contributions, already appreciated by the readers of Codemotion Magazine, can be a tangible support for all professionals in the field who wish to expand their skills, continuously innovate their technical repertoire, and tackle the increasingly complex challenges of the software development world with greater awareness and expertise.

And if, while reading some chapters, you find ideas that contrast with your own thoughts or if you wish to add something, feel free to write to me. This book will be

continuously updated and enriched thanks to your feedback.

# Acknowledgments

I am deeply grateful to my family, whose love and unwavering support have enabled me to complete this project. Without their patience, understanding, and encouragement, reaching this milestone would not have been possible.

A special thanks also goes to my friends who have supported me along the way, offering valuable advice, constructive criticism, and shared moments that enriched my work.

Finally, I want to express my gratitude to everyone who, directly or indirectly, contributed to the creation of this book. Every word, every page, is the result of collective effort and shared passion.

*"Gratitude is not only the memory of the heart but also the light that illuminates the path ahead."* – Anonymous

Thank you from the bottom of my heart.

# Introduction

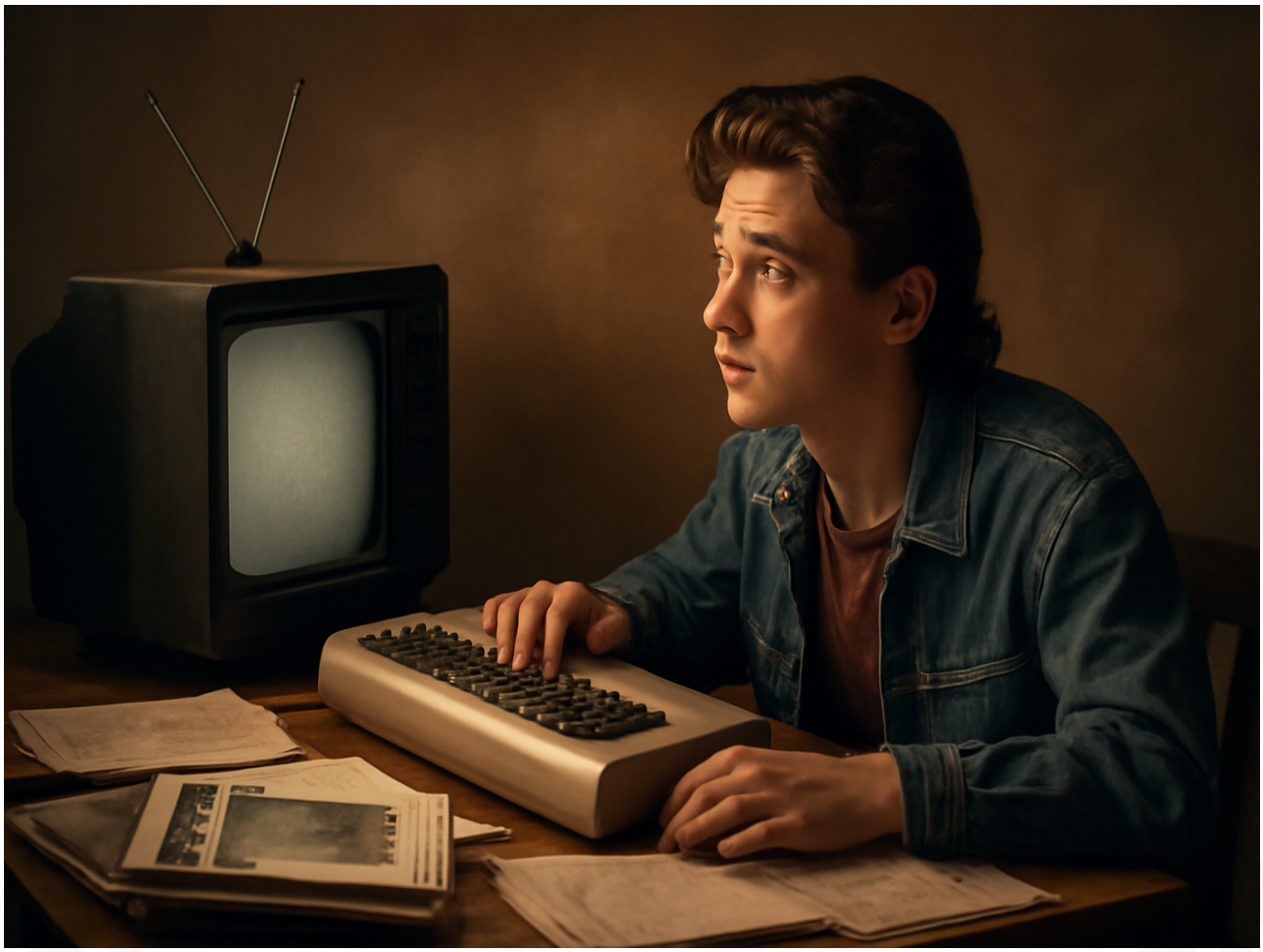
In the fast-paced world of technology, where code is the new universal language, becoming a senior programmer isn't just about years of experience—it's about continuous growth, passion, and strategy. This book aims to transform eager programmers into seasoned professionals capable of tackling the most complex challenges in the software development world.

Each chapter is carefully crafted as a step in a journey of professional transformation. You won't just find technical knowledge here, but a comprehensive roadmap that encompasses technical skills, soft skills, work methodologies, and learning strategies. From enhancing your programming abilities to developing a senior developer mindset, this book will guide you through a unique path of professional growth.

Whether you're a young programmer with a few years of experience or a professional looking to reach the next level of excellence, these pages are designed for you. They are the result of years of field experience, successes, failures, and continuous learning in the software development world.

Prepare for a journey that goes beyond just writing code: you'll learn to think like a true professional, solve complex problems, and build your career with awareness and strategy.

**The world of programming has changed and with it the way to become a programmer**



I still remember the moment I decided to become a programmer: it was 1985, radios were playing "The Wild Boys" by Duran Duran and I had to make an important choice: "Computer or scooter?".

At 14, it's a difficult choice: on one hand you could realize what in your head is the natural evolution of a boy who likes mathematics, that is understanding what computer science is and doing what movies like War Games depicted as something within reach of any kid; on the other hand, buying a scooter, speeding free through the streets, being able to go where you want and showing yourself to girls outside the nerd stereotype.

A difficult choice, agonizing, thought about for months, but faced like all the best choices of my life: instinctively, in a completely random way.

The computer won.

After months of waiting, when I managed to convince my parents too that it was an unmissable choice, I dragged my father to buy it.

My father knew someone who had an "Olivetti store", which in the '80s meant being at the center of the computer world, and the first proposal was an "Olivetti Prodest".

I was young, completely clueless about what programming meant, but I was certain of one thing: the Prodest was the wrong choice. Everyone had the Commodore64, was it possible that I should have a Prodest?

Here too, various days to convince everyone that the Prodest had no future. "The Commodore instead did, it was the computer of the future".

With hindsight I understand how unfounded this statement was, I had no supporting data, except for the hammering advertisements of that period.

I don't remember the day of purchase, I only remember my joy in setting it up: it involved connecting the power and a monitor, which was actually an old black and white TV.

The first day I was devastated: they had given me a game as a gift and I had spent the whole evening playing it.

I thought I would sleep well, but that night I couldn't close my eyes, a mix of joy, euphoria and disappointment. At that age you always have more doubts than certainties: why had I wasted the evening playing, when instead I wanted to be "the programmer"? After all, I didn't even know what it meant, but I was sure it wasn't about playing, but about something different.

In the following days I read the manual several times: a ring binder, in A5 format, with a striped cover and the Commodore logo. I think I still remember the smell of the paper, but maybe it's just an illusion of memory.

From there began years of errors, attempts, presumed successes and failures.

From language to language, from computer to computer, from operating system to operating system, from library to library, from framework to framework.

A journey that lasted decades, where every day I learned something new, where every day I understood that you never stop learning.

But that's the past, a romantic past that will never return: now programming is something different.

## Becoming a programmer today

Becoming a programmer today has completely lost the charm it had in the '80s and '90s.

Once learning a programming language meant having a job, today it only means having a tool that helps you solve part of a problem.

A programming language is no longer a destination, but represents only the first step of a path that has no end.

This aspect discourages many young people from pursuing a programming career and also exposes them to a series of challenges that didn't exist in the past, puts them in a world where more and more often one feels inadequate, in a world where there is fear of not being up to the task.

## The paradox of choice and performance anxiety

Today, an aspiring programmer no longer has to choose between a computer and a scooter, but between dozens of languages, hundreds of frameworks and countless career paths: frontend, backend, mobile, data science, AI, DevOps, and every day the market invents a new acronym where everyone feels unprepared: DevSecOps, FinOps and so on.

As Barry Schwartz observes in his book "The Paradox of Choice" (2004), an excess of options can paradoxically lead to decision paralysis and dissatisfaction. In the programming world, this phenomenon is particularly evident: according to the Stack Overflow Developer Survey, there are over 80 actively used programming languages, with new frameworks constantly emerging.

This abundance is no longer a richness, it doesn't mean we can do anything, rather it's a way to make the programmer feel more and more unprepared: whatever path they choose, the risk of ending up in a niche of little value is increasingly probable.

That's why it's appropriate to increasingly detach from the technical part, specializing in the attitude toward problem solving, which is a practice that has a degree of cross-learning and should help solve the paradox of choosing the best

language.

For those who had a thick tuft on their forehead in the '80s, the question most asked was "how do I make this sprite move?", then it became "which technology will guarantee me a job for the next 5 years?" and now it's starting to transform into "how can I stay current in a constantly evolving world?".

If this isn't enough, think about the pressure created by online communities. If in the '80s comparison was limited to a few friends or magazines, today platforms like GitHub, LinkedIn or Twitter constantly expose the work of thousands of other developers, fueling impostor syndrome.

Most IT professionals experience impostor syndrome at least once in their career, with programmers particularly exposed to this phenomenon due to the collaborative and public nature of their work.

You have the feeling of always being one step behind, you don't have time to read something and think about trying it, when from the basements of a Beijing suburb a team of developers has created a tool that promises to revolutionize the world of development and you, poor programmer who is still being bullied by his work colleagues, feel more and more obsolete.

## The impact of Artificial Intelligence

As if that weren't enough, the advent of AI-based tools like GitHub Copilot, ChatGPT, Gemini or Claude are rewriting the rules.

If on one hand they can accelerate development and help overcome obstacles, on the other they risk creating a generation of programmers who know how to "ask" but not "do". The risk is losing deep understanding of mechanisms, delegating critical thinking to the machine. You learn to use a tool, not to solve a problem from the ground up.

As highlighted in the "State of AI" report, almost all developers in the United States use AI-based coding tools, but only a subset claims to fully understand how these tools generate their solutions.

If we look at OpenRouter statistics on AI usage over time, we can't help but notice a dramatic increase in the adoption of these tools, with exponential growth in queries and interactions.

In fact, AIs have become our daily work companion, an essential tool in any programmer's life.

But the more time passes and their evolution, it becomes increasingly evident that the programmer is destined to get lost in the sea of code they will have to check, uncontrollably increasing the sense of inadequacy we have by nature.

Despite this, Bill Gates states:

```
Programming will remain a 100% human profession, even a century from now
```

A sign that we can probably sleep peacefully.

## What remains (and what's needed) to not get lost?

Yet, despite everything, the heart of programming hasn't changed. It's not the language, it's not the framework, it's not the AI.

The heart is still **curiosity**, the desire to understand, to always invent something new, without limiting ourselves to infinitely repeating what we have learned.

Let's take the example of BitChat, a decentralized messaging application. Its creation required a deep understanding of network protocols and cryptography. The curiosity to explore new technologies and the determination to face new

challenges led to its realization: but how was its core written? Through AI, in a weekend of vibecoding, but without a human mind behind the project, no AI (as of now) would have been able to think it up.

**Logic** capable of breaking down a problem into tiny parts is still an art that AIs must learn, and is the result of years of evolution, which we are trying to emulate day after day.

Another aspect is **tenacity** in the face of a bug that can't be found. A programmer attacks with brute force, then reflects, studies and re-approaches the problem from new points until they find an acceptable solution. Currently AIs only have immense brute force that they try to use to exhaustion, but maybe it's not the best approach for compressed problems or those requiring a holistic product vision.

Perhaps, the real challenge for today's programmer is not learning the latest technology, but learning to manage the noise. Learning to choose a path and follow it with dedication, without being distracted by the frenzy of "new at all costs".

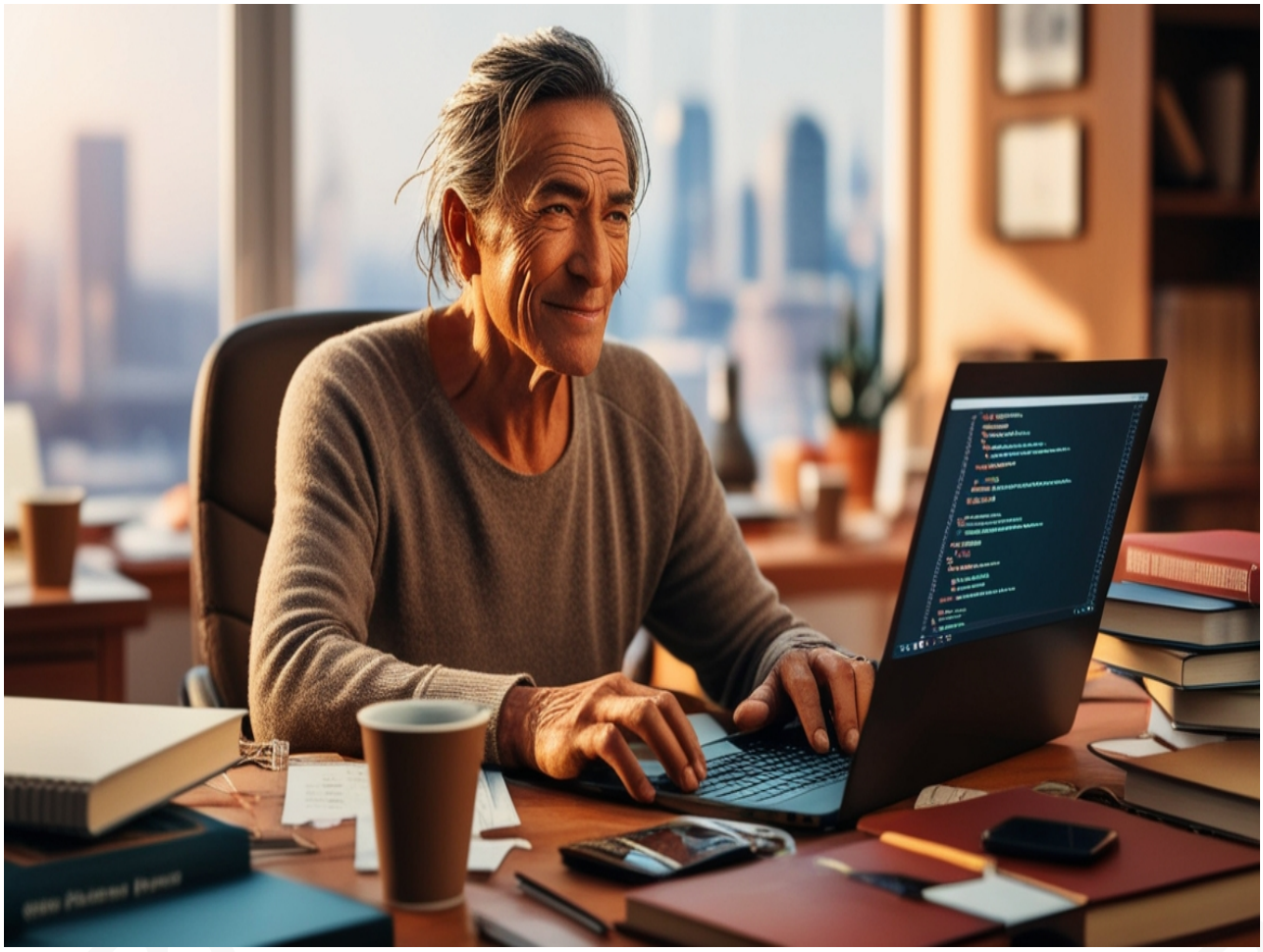
New tools are a help, not a shortcut to avoid thinking, but they must be absorbed to be exploited to the maximum.

To become programmers today, you must first learn to be like that kid from 1985: focused on a goal, curious to discover what's inside the "box" and ready to read the manual, even if today that manual is as big as the entire web.

And thinking back to Bill Gates, let's hope his words don't become a meme like the phrase:

```
640K ought to be enough for anybody!
```

## "Become a Programmer After 40"



Anyone who, like me, spends many hours online, cannot have failed to see this title associated with the sale of programming courses.

It's a title on the verge of clickbait, trying to convince people that, even after the age of forty, one can easily become programmers.

The purpose is clear: given that a programmer's career usually starts very early, the goal is to raise the bar by targeting a group of people who believed they could no longer do it or who perhaps are in the situation of not having a satisfying job and want to do something else: they are looking for the classic professional repositioning.

## Reality

The reason for this trend lies in the growing demand for professionals in the programming sector, where it is currently difficult to find individuals with the necessary skills to meet market demand.

Although it is commendable to try to teach programming to people who are unfamiliar with it, I do not agree with the way in which this concept is being promoted.

To be clear, I am always in favor of programming courses and I view positively those who want to commit and change a situation that does not satisfy them, but I am against messages that lead to the belief that something that requires time and dedication suddenly becomes accessible to everyone.

That phrase in my head sounds like "Become a surgeon after 40" or "Become a lawyer after 40". It's not that it's impossible, but it's difficult, much more difficult than at twenty, especially if you have done something completely different in life.

```
I believe that every individual has the right to choose their
own path in life: one can wake up one day and decide that
programming is their future, just as a programmer can decide to
become a cow breeder.
```

However, let's try to bring a bit of reality into this statement and imagine what will happen at the end of the course that will turn you into a programmer, regardless of the quality of teaching and personal abilities.

Inevitably, it will be necessary to look for a job since trying your luck at freelancing without contacts and experience could be a rather frustrating prospect.

Once you find a company that is willing to invest in you, there could be a series of biases that might change the way potential employers approach you. The first of these biases is certainly age.

## Age Bias

Let's try to consider two scenarios to simplify the reasoning:

If you are 20 years old and you delve into the world of programming, after taking a course or a university path, the expectations on you will be lenient. The lack of experience is normally forgiven, and a great desire to learn is expected. If the company truly believes in training, it is not unusual to also have a mentor willing to help you in your professional growth path.

On the other hand, if you are 40 years old, the expectations towards you are higher, and there could be many biases that are placed before your preparation. Age biases, biases related to your family situation, biases about your learning limits compared to twenty-year-olds: quick to program and with much more time available.

These discrepancies in expectations represent a crucial aspect to consider and for which it is necessary to prepare mentally.

One must be highly motivated if one wishes to embark on this path which, while not impossible, can be really tough at the beginning and might lead to deep

frustration. It's important to keep these aspects in mind and not be discouraged by the initial difficulties, because only over time will one be able to demonstrate one's abilities and make one's experience count.

Ultimately, there is no fixed rule, but it is crucial to be aware of these dynamics to avoid underestimating the challenges that might arise after completing the course and during the job search in the programming sector.

Certainly, one of the mitigating factors that can break down or at least mitigate any preconception is to demonstrate strong motivation, not just in words, but also through tangible actions. Being active on platforms like GitHub, participating in industry events, and collaborating on open-source projects are all elements that demonstrate a concrete commitment and a passion for programming.

These "small pieces" not only testify to the willingness to learn and to put oneself out there, but can also reveal a dormant talent that begins to burst forth at that moment, helping to strengthen credibility and attractiveness in the eyes of potential employers.

## The Training Proposal

I would now like to dwell on another aspect that emerges when reading some training proposals more carefully, I found something that left me perplexed and prompted me to write this article.

A recurring comment, which is found in various forms under this type of path and which I would like to report as an example, is the following:

```
"We teach how to become a software architect, consistently...
to avoid having to start from the bottom."
```

The start of these courses is often focused on "forty-year-old programmers" and just as often tries to sell the concept that people will become "software architects

without starting from the bottom."

I can understand the desire to offer courses to people aspiring to become programmers, and I recognize the existence of a market in which to seek potential talents. However, I remain perplexed by the message that is intended to be conveyed, according to which becoming software architects after a course would not require "starting from the bottom."

It is important to emphasize that experience, or "paying your dues," is fundamental to developing an accurate understanding of designing functional software that can meet the needs of clients and users.

"Paying your dues" implies making mistakes, learning from them, and facing practical challenges in various projects. It means making difficult decisions, sometimes in contrast with best practices, because the context or the needs of the project require it.

Cutting corners on "paying your dues" risks producing projects that, although they may seem well-executed, fail to translate into viable solutions due to issues such as costs, performance, timing, or simply a lack of experience.

If, after completing a course and obtaining the title of "software architect," one wonders why they can't find a job, the answer is simply the lack of "paying your dues." No course can replace practical experience and the learning that comes with it: merely attending a course does not make you a professional programmer or a software architect.

As stated by those who have hit the keyboard more times than me: "There is no shortcut to becoming a professional programmer. You need to face real challenges and learn from mistakes along the way." ("The Pragmatic Programmer: Your Journey to Mastery"). This sentence excellently captures the concept that programming is a skill that can be taught, but it is only through practice that one can aspire to become true professionals.

To raise the level, when talking about the role of the software architect, we are faced with a different discussion.

Being a software architect is not something that can be improvised; it requires years of experience and practice. It is a job that involves the entire structure of a company and can decree its failure or success based on how the work is approached.

Even though a training course can be useful and in-depth, it cannot replace the experience gained over time and will always provide a partial view of the work of a software architect.

Another important aspect is not to confuse the roles of "programmer" and "software architect": they are two distinct roles, each with its own skills and experiences. Being a good programmer does not necessarily imply being a good software architect, and vice versa. However, it is undeniable that a background as a programmer can be advantageous for those aspiring to become a good software architect.

We can draw a parallel with the world of soccer: many successful coaches have had careers as soccer players and have experienced the dynamics of the game firsthand. At the same time, there are cases of players who, in their haste to become coaches, have skipped necessary steps without having the right experience, taking on roles in significant teams and compromising their credibility and career.

While practice is fundamental to becoming good programmers, becoming a software architect requires deep experience and an understanding of the sector that only time can provide. Therefore, it is necessary to be aware of the paths and the time required to achieve the desired professional goals.

## Conclusions

Anyone can become a programmer, at any age, but it's important to be aware of the challenges and expectations that will arise along the way.

The later you start, the harder it will be, but not impossible. Passion and dedication can help overcome the initial difficulties and prove one's worth in the programming field.

After a few years of professional experience, you will understand how crucial it is to tackle the practical learning phase and how illusory it is to believe that a course can replace the importance of "paying your dues" in the programming sector.

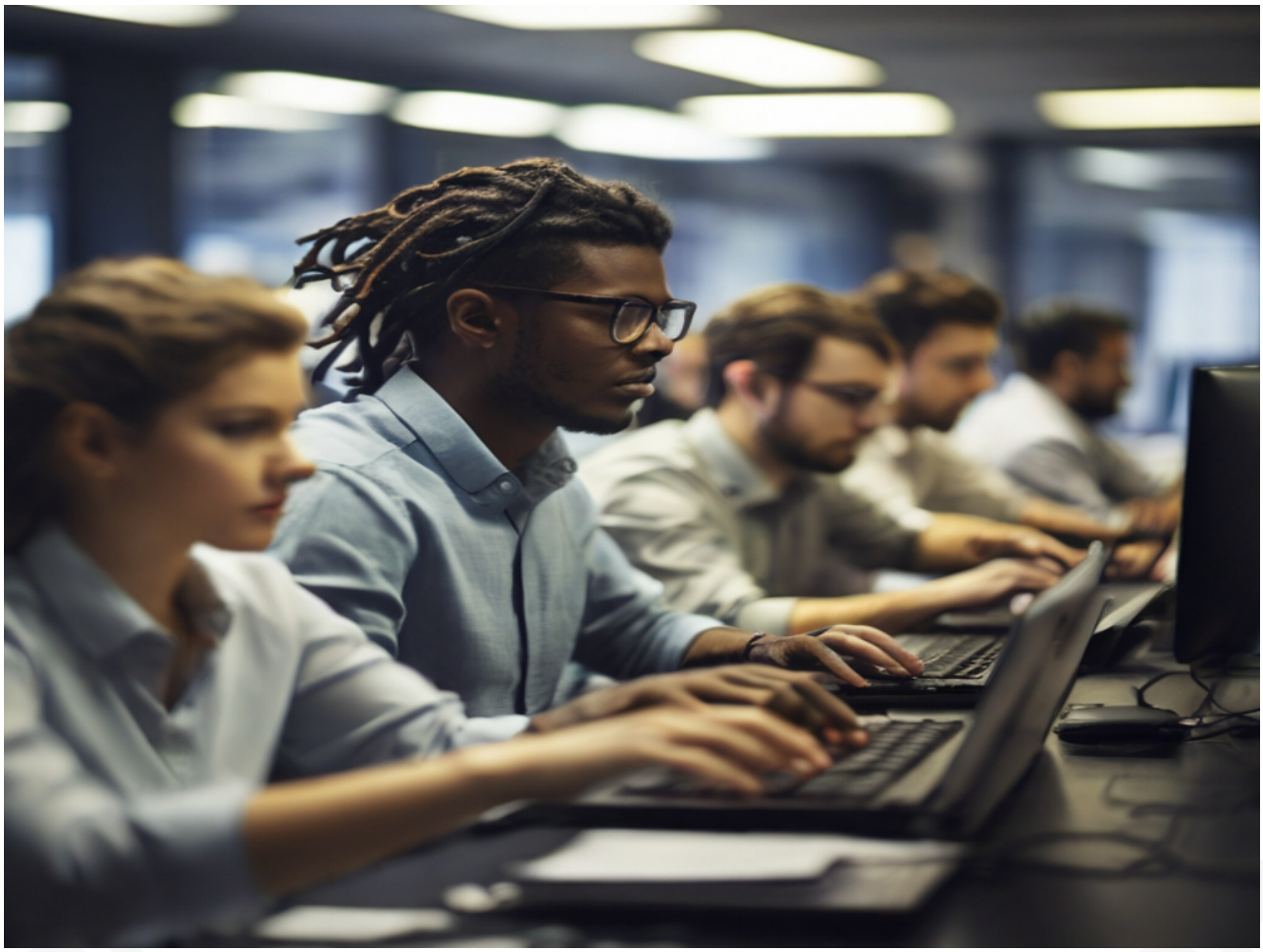
Practice and field experience are indispensable pillars for a successful career in programming. Although courses can provide a solid theoretical foundation, it is only through practical work and continuous research that one can truly acquire the necessary competence and mastery. Therefore, it is essential to recognize the importance of "paying your dues" and to maintain realistic expectations regarding training and professional development paths in the programming sector.

"I took the course, but if I don't practice it, I forget it": will become a phrase you'll often hear and that will make you reflect on the real skills acquired.

## References

The Pragmatic Programmer: Your Journey to Mastery
Can You Become a Programmer After 40?

**Programmers who complete all tasks have not finished their workday**



An individualistic and detrimental approach to managing software projects is emerging among programmers and development teams. The focus of individuals seems to be solely on completing their own tasks as quickly as possible, without considering how this may impact the work of other team members and the fact that this aspect only brings temporary value to the project.

## Completing tasks is not the ultimate goal

This approach may also be the first sign of poor management within the company. It occurs in projects that are heavily structured around tasks, where each person is assigned a task that must be completed as soon as possible, and their performance is evaluated based on this task.

```
That programmer is a war machine, they finish everything
assigned to them in the morning and then move on to something
else.
```

How many times have you heard this phrase and the widespread satisfaction from colleagues and management? But is it really so positive?

To achieve certain performance levels, individuals tend to be isolated, stripped of any concerns, in order to push them to achieve maximum results.

From one perspective, this is positive: all distractions, endless meetings, phone calls, and multitasking situations are eliminated in order to focus on the goal. However, from another perspective, the bigger picture is lost, and there is a risk of creating a toxic work environment that is highly individualistic, where people do not help each other or share their knowledge.

The fact that many programmers are inclined towards isolation when doing their work does not help this situation.

## The racehorse syndrome

You can understand how easy it is, in an environment like this, to not notice the dissatisfaction or disconnect between different team members if you only look at short-term results. Just as a racehorse wears blinders to run fast towards the finish line without looking at the outside world, programmers are driven to complete their tasks.

```
The important thing is the completion of the task, not the
economy of the project.
```

## Rewarding individualism

Thus, individualism tied to tangible results is rewarded.

Unfortunately, a project also consists of a series of intangible results, such as knowledge sharing, collaboration, and personal and professional growth of the people involved.

The metrics dictated by management and many tools used in companies tend to evaluate everything that is tangible: the number of closed tasks, lines of code written, the time between the opening of a problem and its resolution. They rarely take into account intangible tasks.

Think about all the times you have helped a colleague overcome a problem, corrected a line of code that solved a long-standing issue, engaged in pair programming sessions, or shared a tip over coffee.

Therefore, it is necessary to think about projects holistically and try to detach from the logic of the task as the sole objective. A team works together to achieve a common goal. It is not the individuals who win, but the group that is able to continuously and harmoniously evolve and improve the product.

## The problem of "completing tasks"

The real challenge is to transform a group of individuals, operating in an environment that rewards individual performance, into a team that thinks, acts, and works towards long-term goals.

Working as a team is not just a matter of soft skills, but also a matter of company culture and how individuals are evaluated within the company.

Creating isolated compartments benefits no one and must be balanced with activities that bring value to slower or isolated individuals, in order to help them improve culturally and personally.

When teams are composed of people with varying levels of expertise, if you can find a way to transfer knowledge from one person to another, it becomes an enrichment for both the project and the individuals involved.

```
For those managing a software project, it is essential to make
the best use of people's time in order to increase the value of
the project beyond task completion.
```

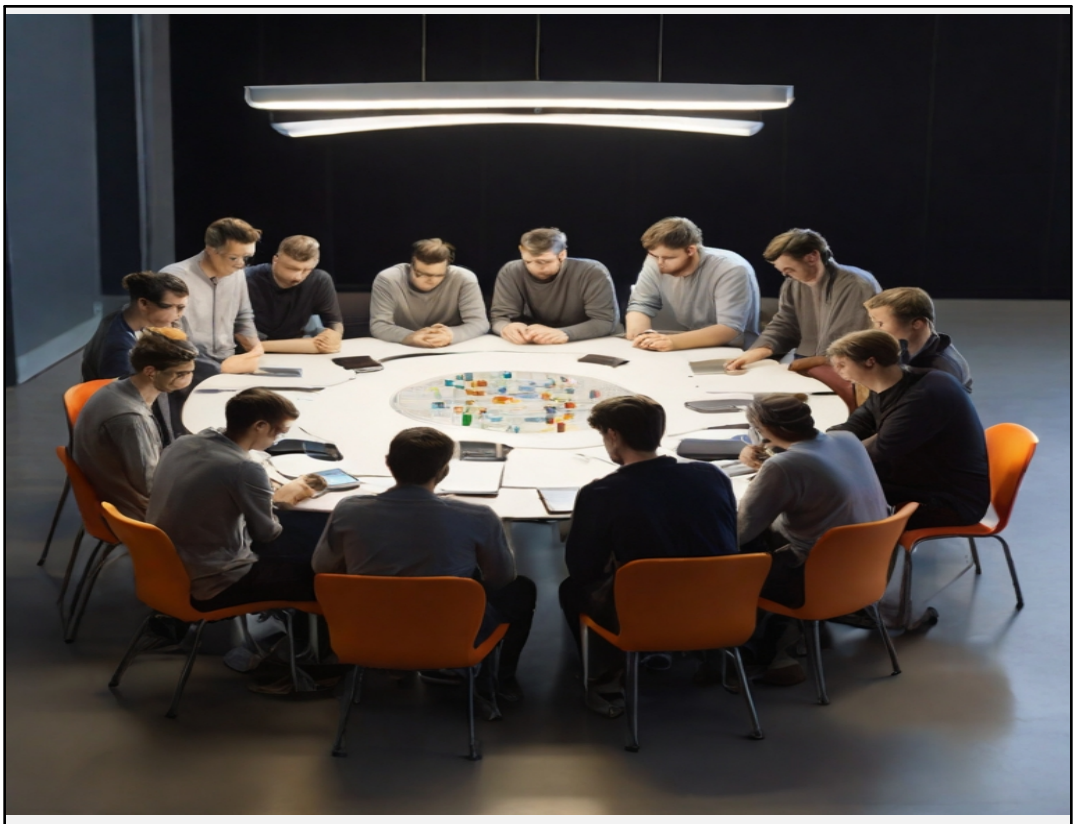
It is essential to promote a collaborative culture within the team, where people can help each other and share knowledge.

## Whose responsibility is it?

Promoting a collaborative approach is certainly the responsibility of management, but individual team members also play an important role in this process. It is crucial that project members are aware that their work is not just about completing their own tasks and going home as soon as possible.

In regular meetings, stand-ups, it is the responsibility of team members to inform others if there are any blocks or if they need help. Similarly, team members must be ready to help colleagues by sharing their knowledge and skills.

However, this aspect may not emerge spontaneously. Therefore, it is necessary for management to actively promote collaboration, identify these situations, and encourage people to work collaboratively.



**regular meetings**

## What strategies can we implement to mitigate this problem?

There are various ways to prevent this approach from compromising the project's economy.

When it comes to soft skills, it is important to include in the team individuals who have a natural predisposition for teamwork, who know how to work in a group, and who are willing to share their knowledge.

If a person completes their tasks and has nothing to do, they could be encouraged

to conduct code reviews of other team members' work, refactor recently added code, or provide mentoring and coaching to less experienced individuals. The performance of individuals should be evaluated based on these types of activities.

## Managing superstars

In a perfect world, everyone can do their job, have lunch at home with their family, and play tennis in the afternoon.

In the real world, we face problems, and not all of them can be solved on our own. Sometimes it is necessary to seek help from so-called "superstar programmers".

In this regard, I am reminded of a speech by Ottavio Bianchi to the Napoli football players when Maradona joined the team:

```
He is Maradona, and we are us. The problem arises if any of us
wants to be Maradona. If we accept this, that he is him and we
are something else, we can win. But if we do not accept it, or
if any of us does not accept it, it doesn't work.
```

Sometimes a superstar is needed to complete certain tasks. Imagine all the situations where you have faced a problem that someone highly knowledgeable on the subject could have solved in minutes, while it took you days.

In these cases, it is easy for the individual to work on well-defined tasks and not need, or have little need for, help from others.

The obsession with completing tasks by the superstar is not a problem, but it is crucial for management to be able to handle these situations and ensure that the work of the "superstar" is understandable and maintainable by all team members.

Therefore, it is essential for the team to subsequently analyze the code written by the "superstar", review it, document it, and refactor it in order to make it

understandable and maintainable by everyone.

## The "assembly line" effect

We must avoid thinking of programming as an assembly line, where each person is responsible for a single task and has no responsibility towards others.

```
Why did you add this line of code that slows down the program?
Because it solves a security issue.
But didn't you realize you could have done it this way?
No, I didn't notice, it wasn't my task.
```

However, be careful not to abuse the opposite approach, where as soon as a person completes their tasks, they are bombarded with continuous assignment of new tasks or the completion of other people's work.

This could lead to situations where people perceive increased stress and an excessive workload, which can prevent them from effectively managing their tasks.

Burnout is a real and tangible problem, and a poorly managed situation could lead to job dissatisfaction and the departure of the best individuals who feel overwhelmed by the number of tasks they have to complete.

Similarly, there may be people who intentionally slow down their work to avoid having to do the work of others. In these cases, it becomes crucial for management to handle these situations, ensure that the workload is evenly distributed among team members, and implement strategies to improve collaboration within the team.

## Conclusions

A good programmer is not someone who completes all assigned tasks, but

someone who is able to work collaboratively and share their knowledge with other team members.

It is necessary to foster the growth of the working group in which one is involved, and it is important for management to not only evaluate their results based on numerical data but also to manage individuals, listen to them, encourage them, and understand where the differences lie in order to bridge them.

The value that a senior brings goes beyond task resolution; it should extend to coaching and mentoring activities in order to help less experienced individuals grow.

This is the best way to ensure that the software project is successful and that the team is able to grow and improve over time.

# The Strength of Admitting You Don't Know

The world of programming is a constantly evolving universe, where technical skills are just the starting point.

Everyone looks at the world of AI with apprehension—software that, on paper, knows more than any programmer and can generate code as easily as taking a sip of water. But the difference between a good programmer and a great one isn't just in the ability to write lines of code. It's also in the ability to correctly interpret specifications, to see beyond what's requested, to recognize when a path is wrong, and to know when it's time to take a different one.

All these skills are acquired through experience, and currently, there are no AIs with such characteristics.

If you call yourself a programmer but still feel afraid, that's normal.

The fear of not knowing enough, of being outdone by software, is a shadow that constantly accompanies every programmer's career.

If you've just learned the difference between a `for` and a `while` loop and break into a sweat when you see a shell, that's perfectly normal. You're a junior, and you're scared of things that one day you'll do with your eyes closed.

But this fear also weighs on the shoulders of those who have been working for years, who have written more `for` loops than their own surname, who can spot bugs just by looking at the code, and who have a holistic view of projects.

## The Fear of Not Knowing

The fear of not knowing is a constant in this job that must be accepted and overcome quickly, as it holds us back, preventing us from growing and improving.

To be a programmer, you need skills, and the more you acquire, the more you

realize there are always new ones to learn. At the same time, you can't stop, limit yourself to a niche, or specialize in a single area, because there will always be some developer, perhaps working from a treehouse, creating code that will become essential for your work.

We can't fully know all the products we use or everything that's released every day: either we dedicate time to understanding what's happening, or we spend our days writing code. But even if we could do both, we still wouldn't be able to know everything.

Who knows everything? The AIs? No, not even them. They learn what they can assimilate from the web, but if the data isn't there or is misinterpreted, the result will be wrong: there's a world of legacy code without documentation that AIs don't know. There are all the closed-source products that can't be analyzed, all those yet to be released, and so on.

So let's make peace with it: we can't know everything, and even with the most modern tools, we'll never be able to know everything. There will always be degrees of ignorance or error in all the code we write.

Similarly, everything we write today might turn out to be wrong: because we don't fully know a product and because technology evolves and changes, and what is considered correct today might be deemed wrong tomorrow.

## I Was Ignorant and Didn't Know It

Recently, I worked on optimizing a Java product that was required to perform a series of operations within a 10-millisecond threshold.

The code seemed correct, but occasionally the program exceeded 100 milliseconds. In an environment where even milliseconds mattered, this fluctuation was intolerable because it exceeded the SLA of the service we had to provide.

What was the problem? Instantiating an object called the class loader to search for a certain type of class instead of directly instantiating a class. This operation, which in 99.999% of cases resolved in less than 1 millisecond, sometimes took up to 100 milliseconds.

The solution? We avoided dynamic class searching, switching to static instantiation. The weight of the change: 1 line of code, but the occasional delay disappeared.

We didn't know the class we were using in such depth, and under normal conditions, the problem didn't exist: it only appeared during moments of high application stress and competition.

How did we figure it out? By talking to each other, comparing notes, and combining our skills and experiences.

## Ignorance as a Resource

In this sea of knowledge—or ignorance, if you prefer to see the glass half empty—the only thing we can do is enhance what are known as soft skills, once underestimated and now more essential than ever.

Let's be clear: technical skills are fundamental, as is the ability to know and use well the new tools made available to us every day. But for those who have spent years working with code, there are skills that make a difference when collaborating in a team.

Given that books, manuals, and AIs will never have all the answers, the thing I appreciate most is the **courage to ask**.

There are skills and experiences that reside only within people, and often it's easier to ask someone who knows rather than trying to figure it out on your own.

For fear of being humiliated or ridiculed, we don't dare ask a colleague a question, we don't say "I didn't understand," we don't have the courage to ask for clarification. We spend hours in front of incomprehensible pieces of code, seeking help anonymously on forums, Reddit, Stack Overflow, or various AIs, but rather than asking a simple question to the colleague sitting next to us, we'd almost prefer to set ourselves on fire. We don't form a group and prefer to bang our heads alone against a problem.

Pride turns into an invisible chain that limits us, paralyzes us, wastes precious time, and leads us to make avoidable mistakes.

We fear being judged as weak, but in reality, admitting you don't know is a great virtue.

Once, an ancient philosopher said:

```
Your mantra for life should be "zero dignity"
```

Actually, it wasn't a philosopher, but Mauro Repetto, founder of 883, but the message is the same: **follow your dreams without worrying about criticism or the judgment of others**.

Many times, it's the fear of judgment that stops us and prevents us from growing. But if we don't ask, how can we learn? If we don't admit we don't know, how can we improve?

## Managing Ignorance

We're all ignorant, but in different subjects. Admitting you don't know and making this aspect clear in a company can sometimes mobilize resources.

Years ago, I was in a company where we decided to focus on C++ for the new product we wanted to create: the reasons were execution speed and the ability to

write code closer to the hardware. Over time, the choice proved wrong, not for the initial assumptions, but because it targeted a pool of programmers who had never worked with that language and came from simpler languages.

In that situation, management noticed a gap: the team I was part of, which was supposed to develop in C++, didn't have, or only partially had, the adequate skills to create the product we had set out to make. For this reason, a course was organized with a language expert to unlock a series of aspects we hadn't considered and didn't know.

Making our ignorance clear allowed us to make a leap forward, helped us understand where we were going wrong, and corrected our course, but most importantly, it made us realize that we're not alone and that asking for help is normal.

If this applies to a team, where it might be easier to ask, it also applies significantly to individuals.

## The Courage to Ask

Years ago, I worked for a company that hired several young people fresh out of school. Their skills varied: some already showed a certain talent, others less so. However, one stood out not for his technical abilities—in fact, he had significant gaps—but for his extraordinary willingness to ask for clarification, even repeatedly throughout the day. This attitude, seemingly simple but actually revolutionary, over time led him to surpass all his peers.

His ChatGPT was his colleagues, the more experienced people who guided him, explained what to do and not do, and where he needed to improve his studies.

The fear of making a fool of oneself is often a cage that keeps us trapped in our comfort zone. We have a choice: remain prisoners of fear or free ourselves with the courage to ask. The path to becoming better programmers starts here.

## The Myth of the Know-It-All

The tech world is full of people who set themselves up as know-it-alls, convinced they know everything and don't need any external help. This dangerous illusion stems from a distorted image of perfection. Consider superheroes as a metaphor: they're portrayed as perfect, infallible, self-sufficient, and fearless beings. But we're not superheroes with superpowers: we're human beings, with all our limits and vulnerabilities.

Try looking at all the people you follow on social media. They often start with topics they know, are very articulate in doing so, and attract followers because of it. Over time, more or less inevitably, the creative vein diminishes, and the frantic search for interesting topics leads them to talk about things they don't know well but think might be of interest.

At this point, the quality deteriorates, but since the audience continues to appreciate it and followers increase, they persist on this path until they find themselves talking about topics they don't know at all.

This is the moment when they become know-it-alls, believing they can talk about everything and know everything, but in reality, they're just bluffing.

This situation happens to almost everyone over time, but it's important to recognize it and not fall into the know-it-all trap.

If your following sees you as a guide, as a point of reference, it's important at some point to stop, understand your limits, admit you don't know, and bring in someone who is an expert on a certain topic.

The difference between a know-it-all and an expert is all here: if we're experts and want the best for a certain topic, we'll go to someone who is more knowledgeable than us on a particular subject and can better guide us, and it's not always an AI,

but a physical person, with a name and surname.

## Asking Alone Isn't Enough

Have we found the courage to ask? To bring in the expert? At this point, a second problem might arise: pretending to understand.

Already asking is a big step, but admitting you didn't understand an answer is a second step that often isn't taken. People prefer to pretend they've understood, even if they haven't, for fear of being judged as stupid.

```
Repeat and rephrase
```

A simple way to verify understanding of a concept is to repeat it and rephrase it in your own words. If you can't do that, it means you haven't understood well and need to ask for further clarification.

## The Difference Between Cooked and Raw Ham

I can't distinguish between cooked and raw ham; it's something I've been carrying for years. My brain refuses to associate the word with the product. I know what I want, but I can't associate the right word.

```
Pass me the cooked ham
```

is one of the most embarrassing questions I can be asked because I know the answer will be:

```
That's raw ham
```

Over time, I understood the problem: I associated the dark color with cooking and the light color with the uncooked state of the food. My brain associated the words the other way around, and I reversed the result, knowing something was wrong

but not understanding what.

After making mistakes repeatedly, I started asking, and after many explanations and ways to remember, I understood how to distinguish cooked from raw ham: but if I hadn't started asking, I might never have had a mechanism in my head to help me understand the difference.

## The Fear of Judgment

Being judged, being seen as someone who doesn't know, is a fear that blocks us and prevents us from growing. But if we don't ask, how can we learn?

Similarly, if we overcome this fear in asking, we must also overcome it in answering: it's not enough to ask; we must also understand the answer and admit we don't understand. Asking without understanding is equivalent to not asking, but also admitting you didn't understand the answer is an art that must be learned over time.

## Fail-Forward Culture

Another concept that fascinates me is the fail-forward culture, or the culture of failure. Failing is normal, it's human, it's a necessary step to grow and improve.

Admitting you don't know, making mistakes, and using failure as a springboard to improve is one of the keys to becoming a better programmer.

Unfortunately, not knowing, failing, showing vulnerability is still perceived as a flaw: in reality, it represents a powerful tool to grow, learn, and improve, a lesson that should be taught from childhood.

```
Don't hold back for fear of making mistakes, but make mistakes
to learn
```

## Conclusions

The next time you hesitate to ask a question, ask yourself: would I rather appear momentarily insecure or remain stuck forever?

And if someone mocks you for your seemingly naive questions, remember that it's they who, by not asking, limit themselves and slowly build an invisible cage from which it will be increasingly difficult to escape.

No one is born knowing everything, but only those who have the courage to admit they don't know can truly learn and grow.

# Beyond Full Stack: Growth Paths in Software Development

The journey to becoming a senior developer is a complex process that goes far beyond the simple acquisition of technical skills. "Beyond Full Stack: Growth Paths in Software Development" explores four fundamental pillars of this professional transformation, starting with the deconstruction of the Full Stack Developer myth - described as a figure who "is neither a backend expert, nor a frontend expert, nor a DevOps expert; knows concepts of UX, UI, SEO, marketing, knows a bit of everything, but nothing in depth" - to the importance of code reviews and software maintenance.

These chapters guide us through an essential professional metamorphosis: from overcoming the illusion of having to master every aspect of software development to the awareness of the consequences of each code modification ("every modification, even the most trivial, can cause performance, regression, security, and maintainability issues").

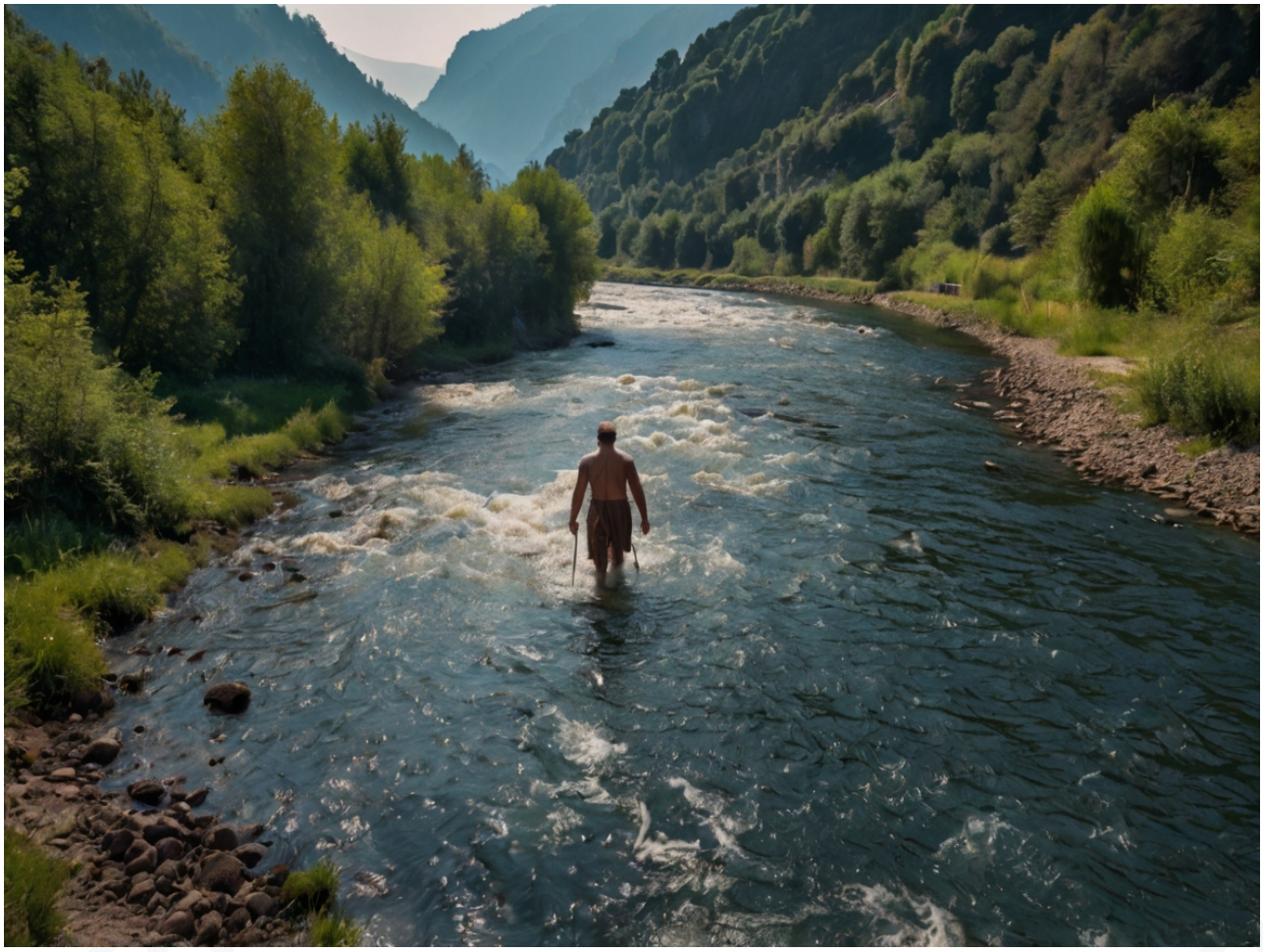
It analyzes the responsible adoption of new frameworks, warning against the risk of "wasting time and resources" on technologies that are not sustainable in the long term, up to the crucial importance of code reviews, where "it's not about imposing one's own style, but finding a compromise that can be accepted by everyone."

You'll discover how the evolution from junior to senior programmer manifests in the ability to critically evaluate each technical decision, from the impact of a single condition in the code ("the loss of carefree attitude and a set of regressive fears") to managing technical debt and removing obsolete code ("maintaining unused code is expensive and over time represents unnecessary technical debt").

Through concrete examples and deep reflections, this text will accompany you in the process of professional maturation, highlighting how each technical decision must be evaluated not only for its immediate impact but also for its future

consequences on the system and the people who will maintain it.

**"Can you change this condition in the code? What does it take?"**



What sets a senior programmer apart from a junior programmer? Many will say it's the ability to assess the consequences of a code change. In reality, it's the loss of carefreeness and a set of regressive fears.

## The fears of seniors

I used to be much quicker at changing an algorithm, modifying a function parameter, creating or destroying flows, and so on. The goal was, first and foremost, to do what the client asked for, and only secondarily to ensure that the code was maintainable or that the choice made was the best among a series of more or less considered options.

Then I aged and started using tests to understand if my code was still good after a change. I began writing documentation to understand why I had reasoned in a certain way, and started doing code reviews to improve code understanding and maintenance.

This new "complicated" approach made my work more complex, or rather: it elongated the processes that, from a request, led to the final solution. A change that I used to make in 10 minutes now takes a whole day because I have to evaluate all the impacts that change can have, run all automated tests, perform some manual tests where automation wasn't possible, write documentation, merge with the main code, wait for the build and integration tests, verify that everything works, and finally declare the issue resolved, hoping I haven't forgotten anything.

Years go by, and it's inevitable that there's an evolution from the carefree child who pushes without tests to the cariatide who also fixes spaces when adding code to the main branch (no, "master" is no longer used).

```
What was once a happy and enthusiastic "yes,"
grows into a "yes, but I also do this,"
slowly becomes a "maybe,"
until it becomes a "no"
```

```
or worse, a "it depends."
```

## Seniors are not bad

It's not out of malice that all this happens, but out of an awareness of what revolves around a single line of code. It's an awareness of the error or impacts of a change and of an extensive regression that has too often undermined the certainty that a single condition can be changed without problems. This doesn't mean that changes shouldn't be made; it just means that impacts should always be evaluated so as not to create bigger problems than the ones you're trying to solve. Often, as one ages, the impacts multiply in the programmer's mind.

Think about the software's lifecycle and its design. Initially, developing a product is relatively easy: we see our goal, have an idea of how to achieve it, ensure that clients and stakeholders are satisfied, and if we're lucky, we manage to create a cohesive team both technically and humanly. This isn't always a given because sometimes just one outlier can destabilize an entire group. Once this goal is achieved, when a change is needed, all team members are aligned and aware of what needs to be done.

As you move away from this phase of collective euphoria and creative phase, the development team starts to change from its original form. At the same time, the client reduces the budget because it enters a maintenance phase, and if due attention isn't paid, mastery of the product begins to slip away.

```
No, it's certainly not the tests that can govern the product,
but the minds that conceived it.
```

Project turnover is an inevitable process, unless you find yourself in an igloo in the middle of the pole, devoid of internet connection, and the poor programmers who developed the project can't even send a resume unless they attach it to a seal (although there's always the possibility that the programmer dedicates themselves to fishing, their great passion suppressed by code).

When a team member changes, a part of the product knowledge is lost, and mastery begins to slip away. Consequently, every intervention becomes more costly, lengthy, and risky.

## Does project maintenance change the rules?

In a normal product lifecycle, some parts require frequent updates, while many other parts, due to maturation, lack of requests, or stability, don't need changes. Making changes infrequently diminishes the programmer's knowledge of the code they wrote. This means that the person who wrote those lines of code two years ago is likely to not remember them and need to review them to recall the processes that led to their creation.

The more vertical the change, the harder it is to remember the reason behind it: you may not remember why you used one parameter over another within a connection to an external resource, why you use one encryption suite over another, or maybe you have an open branch for months, awaiting a client's verification on an urgent feature that suddenly became less important, but now you need to merge it into a baseline with 400 more pushes, and you wonder if it makes more sense to start over or apply the changes to a rebase.

Amidst these types of interventions are the day-to-day tasks, feature changes, bug fixes, requests for new features, bug fixes introduced while fixing a bug, code introduced by a colleague with little product knowledge, behavior changes introduced by an update or a new component that necessitates code changes and introduces an indirect anomaly.

You, who wrote that code, don't remember it. You understand that, in the face of several changes, the modification to be made is larger than anticipated, but you must figure out how to do it because you remember the rationale behind the previous code and understand it's not a simple change. Imagine someone who has to fix the same code and is touching it for the first time.

As Heraclitus said:

```
No man ever steps in the same river twice,
for it's not the same river and he's not the same man.
```

This statement can easily be applied to code: every change, no matter how small, alters the code and the context in which it exists, which is different from the context in which it was created, and the person is also different from the person who wrote that code.

## Use case of a well-known Italian bank

For a moment, imagine being a senior programmer at a "random bank," where you were told to proceed with a system update and related firmware update, assuring that it wouldn't cause any harm and that the update would be done in production. Having seen these types of interventions many times in your life, the first thing that comes to mind is to run a test. However, there isn't a test environment that exactly mirrors the production environment, both in terms of scale and use cases. You advise against the massive update, but the solution provider assures that there won't be any issues. The security department informs you that the update must be implemented by a certain date to comply with company rules and that, in case of problems, a rollback is possible.

Reluctantly, you agree, partly driven by the fact that "we work with agile methodology and must be ready for change" and that "we can't always be negative."

```
Disruptions to online services access (such as app, Invest app,
Internet Banking, and Smart Business)
occurred following the installation of an operating system
update
and related firmware update that led to an unstable situation.
We sincerely apologize and greatly appreciate your
understanding.
```

A "simple" operating system and firmware update led to an unstable situation for a whole 5 days.

Clearly, in this hypothetical scenario, the issue wasn't the update itself but the lack of testing, the absence of a test environment covering production, the lack of an immediate rollback, and the absence of a disaster recovery plan or the underestimation of risk.

## How is the perception of changes outside the project?

Often, those outside the project have a completely different perception of software development and push for changes to be made as soon as possible.

```
I need it yesterday.
```

This phrase is a classic, but often it's not understood that a change, even a simple one, can lead to a series of problems that go well beyond the change itself.

```
I only asked to change one condition, why is Mario causing me
all these problems?
Now I'll assign the task to Bruno, who will solve it in 2
minutes.
```

There are situations within projects that allow code alteration very quickly, and other situations where a change, even a simple one, may require much thought: even if it's just a few lines of code or even a single additional condition. Each programmer approaches the change differently based on their experience, product knowledge, and code understanding. The more experienced programmers are with the product, the more likely they are to carefully evaluate the change because they are aware of the potential consequences. The less a programmer knows about the product, the more likely they are to make changes quickly.

## Conclusions

Every change, no matter how simple, can lead to performance issues, regression, security concerns, maintainability challenges, code understanding difficulties, documentation gaps, testing problems, and deployment issues.

External alterations to the project can create the same problems: I updated the operating system, I updated a driver, I updated the firmware, and now nothing works.

The pressure on a programmer increases progressively over time because awareness grows of everything that can go wrong and how difficult it is to solve a problem once it occurs.

This can lead to situations where a senior programmer refuses to make a change because they know the risks are too high or because they know the change would require too much time and resources to be done correctly.

The next time you look at a programmer who has been working on the same code for many years and doesn't want to modify it, and warns you about every internal or external alteration to the project, don't think of them as incapable, but rather as an aware individual. Use their advice to avoid project collapse and weigh the risks and benefits of each change carefully because, as Isaac Asimov said:

```
No sensible decision can be made any longer without taking into
account
not only the world as it is, but the world as it will be.
```

## Adopting New Frameworks Could Jeopardize Your Project

In the world of software development, the allure of new technologies is always strong. It's tempting to get your hands on a new product, especially one that promises to solve a series of problems that other frameworks can't address.

These new projects often boast a small but energetic community, promise to speed up development times, and offer interesting innovations compared to their competitors. All of this sounds very promising, but is it really the right choice for a long-term project?

Donald Knuth, one of the fathers of modern computing, commented on his own code this way:

```
Beware of bugs in the above code; I have only proved it
correct, not tested it.
```

If this had been written by a junior programmer, it might seem like a joke. But coming from Knuth, it's not only humorous but might also indicate that the code was syntactically correct but not testable.

Sometimes, it's much easier to demonstrate a concept, describe its functionality, and document it, rather than actually applying it, which might involve a series of unforeseen aspects.

This concept can be paraphrased as: "A brilliant idea doesn't always work in reality." Thinking about this, I can't help but consider the world of blockchain: excellent tools on paper, but often criticized in practice for performance and resource usage.

Stepping outside the software world, think about products like the Segway, which, despite being a brilliant idea, didn't achieve the expected success. Or Google Glass, which, despite being an innovative product, never took off.

## And the Frameworks?

Frameworks and, more generally, software technologies often suffer from initial hype that excites programmers, distracting them from a solid analysis of their long-term effectiveness. Are the concepts used in that new software applicable to all projects? Are they easily integrable with other technologies? Are there solid companies investing in that technology? Is the community active and supportive?

The so-called "early adopters" are often attracted to novelties but frequently don't realize the risks involved in this choice. If everything goes well, they've made a lucky choice that leads to the success of their software. But if the chosen product fails to deliver on its promises, they end up with software that doesn't work as it should and requires much more maintenance and refactoring than anticipated.

For those who have been developing software for over 30 years, the name Visual Objects might ring a bell: it was supposed to be the new Clipper and bring millions of DOS programmers into the Windows environment. A fantastic idea, but it didn't work: too many problems, too many bugs, too many limitations. At the time, it had the backing of a company like Computer Associates, which could afford to invest millions of dollars in a product, but despite this, it never took off.

That's why, when it comes to choosing a framework or technology for a project, it's important to consider not only its technical features but also its long-term sustainability.

Frameworks

## The Time Horizon: A Crucial Dimension

Whenever I evaluate a project, I try to anticipate its lifecycle, or at least what it should be, based on my experience and the information at my disposal. The crystal ball doesn't exist, and the future is unpredictable, but there are signs that indicate whether a project will be short, medium, or long-term.

To understand how long a project might last, it's important to consider various aspects:

• **Long-term objectives**: What are the project's goals? Is it a product that solves a single problem or something structured to last years or just a few months?

• **Market stability**: Is the market in which the project operates stable or rapidly

evolving? The dynamism of a market can lead to rapid changes that require greater flexibility from the project.

• **Competition**: What is the level of competition in the sector in which the project operates? Working in an arena where other players might come from larger, more structured companies can require greater attention to the quality and sustainability of the project.

• **Industry trends**: What are the current trends in the sector in which the project operates? Some projects are born outdated because they are based on obsolete technologies or outdated business models.

• **Available resources**: What resources are available for the development and maintenance of the project? Are these "one-man show" projects, or are there structured teams behind them?

These are just some of the possible indicators, but relying solely on the name and what is promised often underestimates the risk.

When using young frameworks or technologies, it's important to consider that they might not be supported long-term or might not adapt to market changes. This can lead to high maintenance costs, compatibility issues, and increased code complexity.

On the other hand, it's also necessary to evaluate the projects you intend to undertake. They might be projects born and completed over a weekend: in this case, the primary need is problem-solving. Others might need to last a few months to meet a specific need. Finally, there are projects destined to last years, on which a company intends to build its operational base: in these situations, the choice of framework and project architecture becomes crucial and cannot be based solely on the trend of the moment.

The time horizon is a dimension that every software architect should consider because it significantly influences the expectations and choices that can be made.

For a "one-shot" project, you can safely adopt the CIRO framework, created by a nomad from Tanzania, used only by his group of friends, with a single release made a few years ago, if it astonishingly solves a crucial problem for the project. If, however, I need to create something that must last beyond the expiration date of a dessert, perhaps I should moderate my aggressiveness in choosing the software components to use and base my decisions on different KPIs.

## The Risk of Abandonment

I've used both "closed source" and "open source" products whose creators disappeared into thin air, as did the community that was part of it. If this happens with "closed" products, it's certainly very worrying, but don't fall into the illusion that everything is rosy if it happens with "open" products. Using a product is one thing; developing it and knowing it inside out is another.

The mind of someone who designs a solution is usually trapped between the lines of code of the product itself and is certainly more immersed in its internal logic than someone who uses it, who often exploits only a part of its potential. So don't think that "open" means "there's a problem: I'll fix it," because it's not always the case.

Linus Torvalds, creator of Linux, has a strong opinion on this subject:

```
Software is like sex: it's better when it's free.
```

Although Torvalds uses this quote to promote open-source software, it's important to remember that "free" doesn't necessarily mean "without costs" and that the freedom to access a project's source code doesn't necessarily mean being able to maintain and evolve it.

## The Importance of Periodic Analysis

This is one of the reasons why we should always perform an analysis of the life of

the products we adopt. An analysis to be repeated periodically, taking into account the current context, the project's maturity, and its future prospects.

If this aspect can be ignored for short projects, for those that exceed years of development, the problem becomes pressing: either the product and its dependencies are updated, or you find yourself in a dead end due to the countless entanglements of your code.

A periodic analysis allows us to evaluate whether the framework or technology we are using is still suitable for our needs, if it is actively supported by the community, and if it can adapt to market changes.

## The Risk of Non-Standard Services

This also applies to all the problems that can arise from adopting a non-standard service, which could be withdrawn from the market or double its prices from one season to the next.

Jeff Bezos has an interesting view on how companies should approach technology:

```
You have to be stubborn on the vision and flexible on the
details.
```

This philosophy can also be applied to the choice of frameworks, services, and software development: while technical details can change, fundamental principles like scalability, maintainability, and interoperability should remain constant.

For this reason, creating software that is tightly coupled with a framework could lead to maintenance and scalability issues in the future. If the framework is no longer supported or cannot adapt to market changes, we might find ourselves in a situation where we need to rewrite much of the code to adapt it to a new technology.

I've seen products make technical choices that, on paper, seemed the best, only to realize over time that the software coupling was so strong that changing one part meant having to change everything else. This problem can also be avoided with proper analysis and a thoughtful choice of components used, decoupling usage from implementation and accepting the need to change part of the software if necessary.

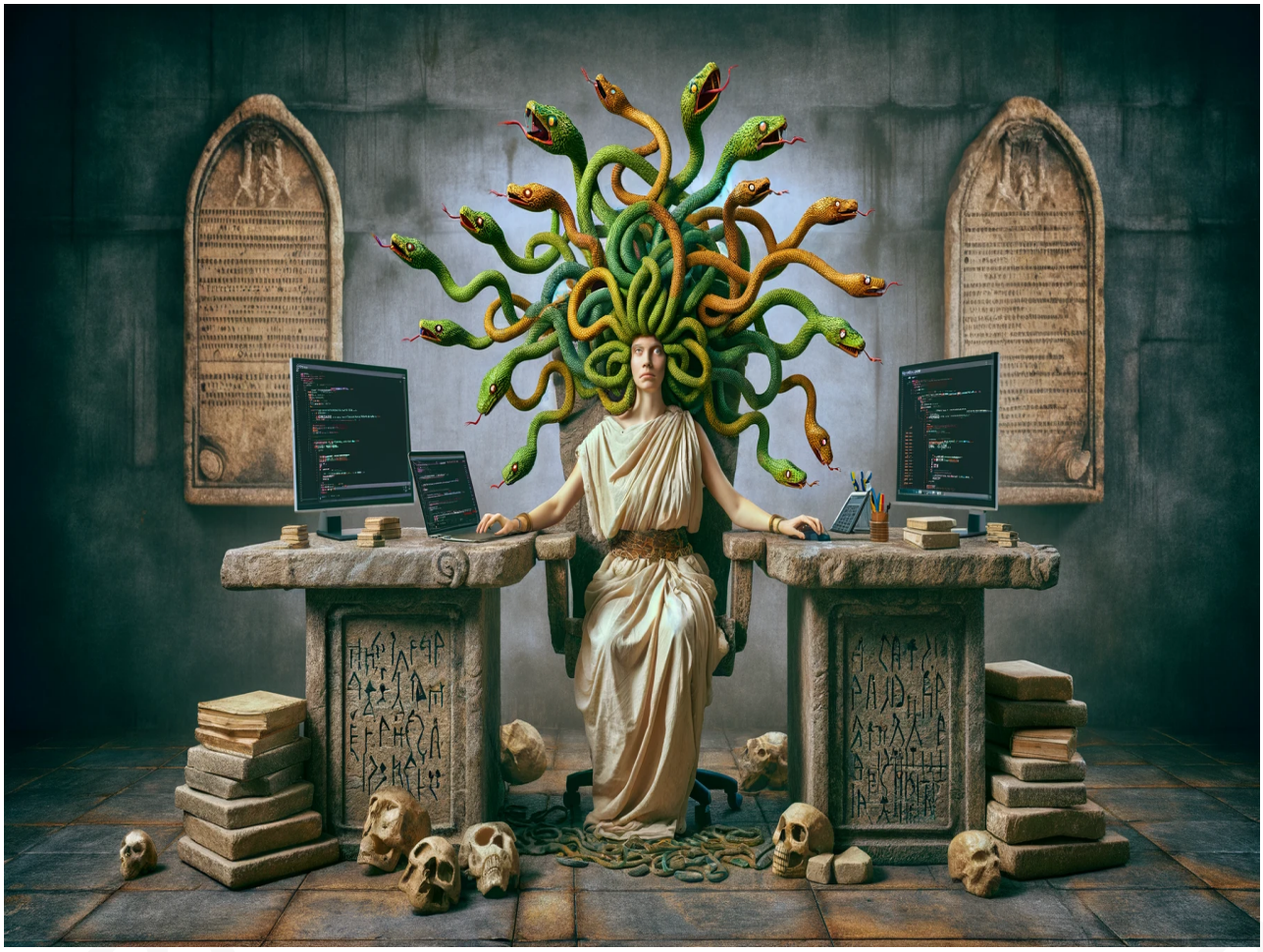
## Conclusion

When it comes to young solutions or frameworks, the risk of wasting time and resources is considerable. If you love uncertainty and the project's duration is not a factor you want to consider, you are free to make any choice and build a product full of unanswered questions.

In case of errors, keep in mind that you might need to start from scratch unless your software is sufficiently modular to allow the replacement of a component without having to rewrite everything else.

The choice of a framework or technology should be guided not only by enthusiasm for innovation but also by a careful evaluation of long-term risks and benefits. Sustainability, maintainability, and scalability should always be at the top of our priorities when making architectural decisions.

# The Myth of the Full Stack Developer: an Uncomfortable Reality



The term "Full Stack Developer" has undergone a radical transformation that deserves careful reflection.

Let's start with the definition of what it means to be a "Full Stack Developer," drawing from ChatGPT:

```
"Full Stack Developers" are software developers who have skills
in both front-end and back-end web development or software
application development. In other words, they are capable of
working on all parts of an application or website, from the
client-side that runs in the user's browser (front-end) to the
server-side (back-end) that handles business logic and data
access.
```

## My Approach to the Term Full Stack Developer

At a certain point in my career, this definition perfectly reflected my state of mind. It was what I felt I was and what I wanted others to perceive when they looked at me. Proud of this status, I included it in my resume, almost like when you get an excellent grade in school and want to share it with your family.

Over time, I realized that the allure of this definition had captured the hearts of many people, mainly due to a common belief: being Full Stack means being a complete programmer, and for companies, hiring a Full Stack Developer represents added value.

If you assign a project to this type of programmer, they are capable of independently handling all aspects of software development. It's like having an entire development team encapsulated in one person!

Over time, I realized that this approach could make sense when projects were simple and technologies were few.

Today, the world has changed, and it is much more complex to cover every facet

of a project. Now, being a "Full Stack Developer" seems to be synonymous with working mediocrely on all aspects of a project: you are not an expert in backend, frontend, or devops; you have a basic understanding of UX, UI, SEO, marketing, you know a little bit of everything, but nothing in depth, or maybe just one aspect.

Let's try to translate the concept of "Full Stack" to your mechanic: your mechanic, when there is a problem with the bodywork, doesn't fix it by hammering it but involves a bodywork specialist; when there are carburetion problems, they call in a carburetor expert; when there is an electrical problem, they involve an auto electrician.

Similarly, if your doctor suspects a particular problem, they send you to a specialized expert in that area, who will conduct a thorough examination to formulate a targeted diagnosis and treatment.

A "Full Stack" programmer, on the other hand, is chosen not to have to call anyone, to solve everything on their own.

This approach is not sustainable, or rather, it is not feasible if we think of Full Stack as an endpoint rather than a starting point. Many professions make us understand that an expert in a particular field is preferable to a "jack of all trades" if we want to do a job with artistry.

Experienced programmers know how important it is to specialize and recognize the hours wasted searching for solutions that an expert would have quickly resolved.

In mass culture, and often when talking to clients, there is the presumption that a programmer should be able to do everything perfectly.

Responding with phrases like "it's not my field, we would need a UI expert" is seen as a sign of weakness, even though it is a legitimate, professional, and honest response.

```
- "I am a Full Stack Developer"
- "Ah, so you're an expert in everything?"
- "Not exactly"
```

## The Job Market and the Full Stack Developer

The job market has started riding the wave of demand for "Full Stack Developers," responding to this need in sometimes bizarre ways.

Figures like "Junior Full Stack Developers" are emerging, who declare themselves "full stack" but add the term "junior" to emphasize their inexperience. Often, this title follows intensive courses like "become a full stack developer in 6 months".

We may not realize it, but we are facing a serious problem. On the one hand, we are putting in the heads of young programmers the idea that they can independently and easily cover the entire technological stack, only to have them face the harsh reality: when they encounter their first complex project, they realize they are not able to handle it alone.

However, it is not entirely the fault of these new programmers. There is also a problem related to some companies that do not have the ability to accurately assess the real level of competence of individuals and assign roles of responsibility based on limited information.

Responding brilliantly to a technical interview is one thing, but as Linus Torvalds wisely asserts:

```
Talk is cheap. Show me the code.
```

## Code Quality and Programmer Evaluation

The code, the approach to problem-solving, and how difficulties are tackled are the aspects that distinguish one programmer from another, even if both define

themselves as "Full Stack Developers".

Unfortunately, evaluating how capable a person is of producing quality work is not easy at all. I have seen programmers produce large amounts of code but of poor quality, and programmers who produce less but of extremely high quality.

To quote Antoine de Saint-Exupéry:

```
Perfection is achieved, not when there is nothing more to add,
but when there is nothing left to take away.
```

Being able to assess the true value of a programmer is a very difficult skill to acquire, yet it is a fundamental competence for a good manager. It is not enough to evaluate the number of lines of code produced, the number of completed projects, or the number of technologies known, especially in an era where technologies change rapidly and help us write more code in less time.

The amount of code is not an indicator of quality, even if it works perfectly and is covered by numerous functioning tests. Code should not be evaluated by weight but by its quality, maintainability, scalability, and readability.

Who can evaluate the quality of the code? It is not easy to answer this question.

The simplest answer might be another Full Stack Developer, but that is not the right answer.

It is not the right answer because a Full Stack Developer is not an expert in everything but should be an expert in their specific application domain. Therefore, the evaluation of a person's value by a Full Stack Developer may be influenced by a series of biases due to their personal experiences, previous knowledge, and acquired skills.

So, can an HR person evaluate the quality of the code? Or a manager? A

colleague? A client? The right answer is: all of them can contribute.

Even in this case, however, the evaluation will be the sum of the experiences of the various people involved, who may have different opinions and decree that the same Full Stack Developer is a genius in one technology stack like LAMP but incompetent in another like MEAN.

## What if We Focused on "Problem Solving Attitude" Instead?

We have now understood that "Full Stack Developer" is a non-qualifying term, open to interpretation depending on the context, and, above all, very burdensome in terms of the necessary technological updates.

Let's try working on a different concept: "problem-solving attitude".

A programmer who possesses a strong problem-solving attitude is capable of tackling any problem, even if they have never encountered a similar one before. They can analyze the problem, break it down into smaller parts, find a solution, implement it, and successfully test it.

This type of programmer is not discouraged by difficulties but faces them with determination, adapting and learning along the way. This flexible and analytical mindset is crucial for tackling the ever-changing challenges that arise in software development.

A programmer with a strong problem-solving attitude is not afraid to ask for help, to admit "I don't know" or "I can't do it". On the contrary, they are capable of working effectively in a team, sharing their knowledge, learning from those who know more, and teaching those who know less.

In this context, a wealth of experience in different technologies and contexts, in international projects with heterogeneous technologies, is undoubtedly an added value, much more than being a Full Stack Developer. The latter does not specialize

in problem-solving but in an in-depth knowledge of their specific technological stack, which is out of context like a bowler in a game of briscola.

Being able to solve problems even in unfamiliar contexts is a valuable asset within a team. There will always be time later to refine the solution by placing it in the hands of a product expert, but the ability to solve problems independently is an invaluable skill.

A programmer with a flexible, analytical, and collaborative mindset, combined with a wide range of diverse experiences, will be able to successfully tackle the most complex challenges, adapt, and constantly grow.

## Conclusion

My advice is to avoid declaring oneself a "Full Stack Developer".

This term is now too inflated, and when written on the resume of a junior programmer, it adds no value, except for the amusement of more experienced programmers who read it.

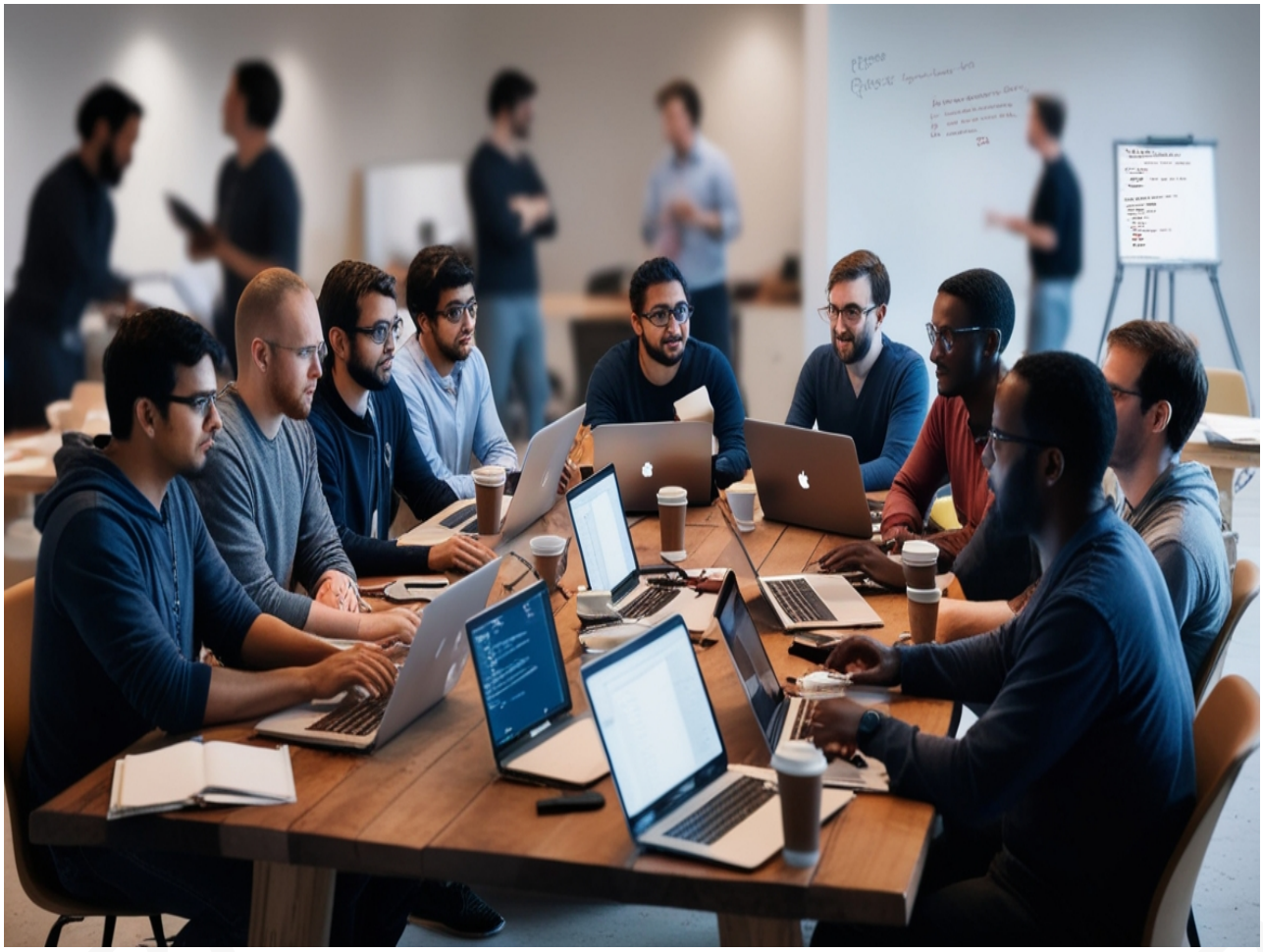
It is much better to accurately identify the technologies in which you are an expert and those in which you have less experience. This applies to all programmers: you can be a wizard in Angular, have written REST APIs in Node, but if a backend expert analyzes your code, they will probably find gaps.

By clearly stating your skills and experiences, it will be up to your interlocutor to evaluate your actual value, which is much more important than a generic "Full Stack Developer" label.

We should focus on building a solid problem-solving attitude, gaining experience in different technological contexts, stepping out of our comfort zone, and promoting a collaborative approach to work. These qualities will be much more appreciated than self-certifying as "Full Stack," which risks being misleading or

excessively generalist.

## "Let's See Who's Got the Biggest One: Let's Do a Code Review"



Code review is one of the most delicate and important phases of a software project. Any software project we've developed or will develop over time has been or will be subjected to a code review.

It's unimaginable for code written by anyone not to undergo changes over time. While this might apply to individual parts of a project, when we look at the entirety of a software project, code review becomes a significant step between staying small and playing with dolls, or stepping out and becoming adults.

This step is therefore crucial to ensure code quality, to prevent errors from compromising the software's functionality, and to broaden the team's understanding of the code. Yes, because evolving the team is more important than indulging one's ego by looking at a few lines of code that only the author can understand.

## Code Review is Annoying

However, there's a fundamental problem: programmers find code review annoying, they don't like it, they consider it unnecessary or wrong if done by others, and believe it should be done and decided only by them.

For some, it's like losing a child:

```
That code was written that way for a very specific reason:
Mario didn't want anyone to touch it, but now that he's left,
no one understands it anymore.
```

A good time to do a code review is when fixing a bug. However, it's necessary to break away from the approach often adopted by programmers, who tend to be surgical and address the single issue without leveraging holistic thinking.

```
I've been told there's an error when clicking a button, so I'll
make sure there isn't.
```

In reality, the problem could be much deeper and require broader reflection. We should question why that button was clicked, if it was clicked by a user, if it was clicked by a user who shouldn't have clicked it, if it was clicked by a user who shouldn't have clicked it at that time, if it was clicked by a user who shouldn't have clicked it at that time and in that context.

Expanding the thought, we might also ask if the problem extends to other buttons, how and if it's reported, if it makes sense to have that button or those buttons performing that operation, if the operation that button performs is necessary, if the operation that button performs is necessary in that context.

Broadening the context, however, means spending time, using resources, questioning one's work and that of others, questioning one's ego and that of others, but by doing so, the code improves, the system improves, the team improves.

Therefore, it's necessary not to look at the single issue reported to us, but to try to understand the context in which it was made and if there are other parts of the code that could be improved.

Holistic thinking, even in code, is important: if we don't understand that a single change impacts the entire system, we can't understand the system, and if we don't understand the system, we can't improve it.

As Plato said:

```
Refactoring is the art of removing the bad and adding the good
to the code.
```

Perhaps the author wasn't Plato, but the concept is clear: the code needs to be improved, made more readable, more maintainable, more testable.

## Unused Code

If you've worked on the same project for more than five years, the phrase:

```
We wrote that code for an "Important Client," who is no longer
our client, but we don't remove it because it "might" be
needed.
```

is almost a mantra that I'm sure you've heard at least once.

Layering thoughts like this over time leads to having code that is incomprehensible, whose purpose is unknown, and whose workings are unclear: the specifications were intertwined with the client's needs, the team's requirements, and the technical choices of the moment.

Meanwhile, the code has changed, the team has changed, the client has changed, the needs have changed, the technical choices have changed: it's right for the code to change or be removed from the project. Maintaining unused code is costly and over time represents an unnecessary technical debt.

Some companies have realized how costly and pointless it is to maintain projects that are no longer used: code is a cost, not a value, if it's not used.

Google, for example, has a rich history of projects that have been abandoned. You can find traces of it on Wikipedia:

```
https://en.wikipedia.org/wiki/Category:Discontinued_Google_serv
ices
```

## I Use Tabs, Those Who Use Spaces Aren't Good Programmers

In an episode of "Silicon Valley," the protagonist Richard discovers that his girlfriend Winnie uses spaces instead of tabs to indent code and breaks up with her: it's a cinematic exaggeration, but I know people who would never agree to work with someone who uses spaces instead of tabs and vice versa.

Breaking this habit, and this is coming from someone who has used spaces since they first touched a keyboard, is difficult but necessary: the code must be uniform, consistent, and readable. Having different styles within the team is counterproductive, slows down work, increases the possibility of errors, and makes maintenance difficult.

Within a code review, it's necessary to discuss these things too: it's not about imposing one's style, but finding a compromise that can be accepted by everyone and hopefully isn't a burden for anyone.

Once an agreement is reached, it's important to respect it: if it's decided to use spaces instead of tabs, you can't go back.

However, this is something difficult to accept and implement, both because some programmers consider code like a child that can't be changed, and because even when imposing a uniform style, sooner or later someone won't understand the specification, others will configure the IDE incorrectly, or lose settings after the first update.

In these situations, where the language itself doesn't help us maintain a uniform style (yes, there are languages that do this for us in an absolutely dictatorial way), it's important to use tools that allow us to automate code formatting as much as possible.

In one of the largest projects I follow, it was decided to adopt OpenRewrite, a tool that automatically reformats code in a uniform and consistent way, reducing differences between different programming styles: standardizing not only makes the code more consistent but also speeds up the onboarding of new team members and the transfer of a component from one team to another.

## Resistance to Change

Even though most programmers agree on the need for a code review, they often

refuse to accept proposed changes, partly out of laziness, partly out of pride, and partly out of habit.

If you've been working the same way for years, and it works, why change?

```
My code is born perfect
```

Unfortunately, no: it's not like that. Within a team, the aspect that must prevail is the readability of the code by everyone, the ease of onboarding a new resource, and the simplicity of modification even months after not touching certain lines.

It doesn't matter if the IDE you use formats the code in a certain way, it doesn't matter if you've always liked declaring variables with "_", adhering to a widespread standard, spontaneously or through automation, is an advantage, even if at first it may seem like a burden.

## Clean Code

One of the goals a code review should have is to write clean, readable, and maintainable code.

Sometimes I've faced the choice of whether to keep code that worked or dismantle it because, despite passing any functional test excellently, it was written in an obscure, difficult-to-read, and maintain way, and every time I put someone in front of that code, their expression was always the same:

```
I don't understand what it does
```

Sometimes we believe that using the latest syntax proposed by a language is the solution to all our problems, but that's not always the case. Sometimes new constructs are based on concepts that are difficult to understand and explain, sometimes they're just sugar to make us feel more skilled: "sugar code" as some call it.

Not always is the most compact and concise code the best: sometimes it's better to write 10 lines of code that do what they need to do clearly and legibly, rather than a single line that does the same thing but that no one understands, gaining nothing in terms of performance and functionality.

## Automate What You Can

Strangely, programmers are more willing to accept something proposed by a program than something proposed by someone within the team.

If your team has the same atmosphere, you can propose introducing, within the project pipeline (hoping you have one), processes for code normalization and static analysis.

The former will help reduce style differences between programmers, while the latter will suggest changes, help us understand where bugs are hidden, where tests are missing, where cognitive complexity is too high, or where we can reduce code without losing clarity or effectiveness.

## But I Don't Care

Sometimes I spend my free time studying projects by other programmers, running some analyzers, and trying to fix some code: it's good practice and allows me to see how others tackle problems or underestimate the consequences of their code.

During these forays, I've encountered completely different approaches from individual programmers or teams of programmers to whom I submitted my Pull Requests.

Among all the projects I've analyzed, I'd like to talk about two that particularly struck me for how the team handled my contribution.

The first Pull Request concerns the JDK of OpenJDK. I noticed that, by mistake, 82 sources contained double ";" at the end of a line.

You understand that it's not something important, we can even say it's something negligible, but I decided to make a Pull Request to fix the issue and see how the development team would react.

You can find the PR at this address:

```
https://github.com/openjdk/jdk/commit/ccad39237ab860c5c5579537f740177e3f1adcc9
```

The approach, in my opinion, was interesting: first of all, 8 people were involved in analyzing the change, as the changes were horizontal across many Java packages.

After a horizontal discussion among various maintainers and realizing that the issue was real, an issue was opened:

```
https://bugs.openjdk.org/browse/JDK-8282657
```

There was also a discussion about a possible change to the build tools, which already removed spaces at the end of lines, introducing the removal of double ";" characters.

This approach denotes a team that analyzes every single change, discusses it, and tries to understand if the proposed change is actually useful or just someone's whim.

In this case, they understood it was a trivial change but aimed at code cleanliness, and for this reason, it was accepted.

A second case involves a security tool developed by a single maintainer, also on GitHub.

In that case, I had proposed a much more serious PR. There was redundant code, synchronized classes were used in processes where synchronization wasn't needed, some resources were opened without explicit closure, and so on.

It was therefore a change aimed at improving code quality and reducing the possibility of errors, not something trivial like removing a character.

Moreover, switching to unsynchronized objects led to a performance increase of around 20%: all this without distorting the code, but simply using the right constructs and, with equal interfaces, the right classes.

In this case, I was convinced there would be no problems integrating this code into the project, but the response was negative.

The maintainer replied that he didn't intend to integrate suggestions from a syntactic analyzer into his code and that I could use them on my fork, but he would never integrate them.

```
If you don't like my code, fork it and modify it as you wish
```

This is the classic response of someone who doesn't want to change, who doesn't want to accept that their code can be improved, who doesn't want to accept that their code can be changed by others, even though the changes can improve readability and lead to tangible improvements.

## Conclusion

Even when everything works, reviewing code is important: it allows for project improvement, team improvement, and cutting down on technical debt.

Reducing technical debt, having the courage to dismantle even working code, discarding unused code, listening to static reviewers, standardizing programming style, automating code formatting as much as possible, are all steps that can improve the code and the team.

Don't be afraid to question what was written in the past and broaden your vision when you have to work on the code: don't just look at the single change, but try to understand the context in which it was made and if there are other parts of the code that could be improved.

A better product comes through many small steps, and even with hundreds of working tests, the code isn't necessarily perfect: code review is a fundamental step to ensure code quality and prevent errors from compromising the software's functionality.

# Professional Dimension

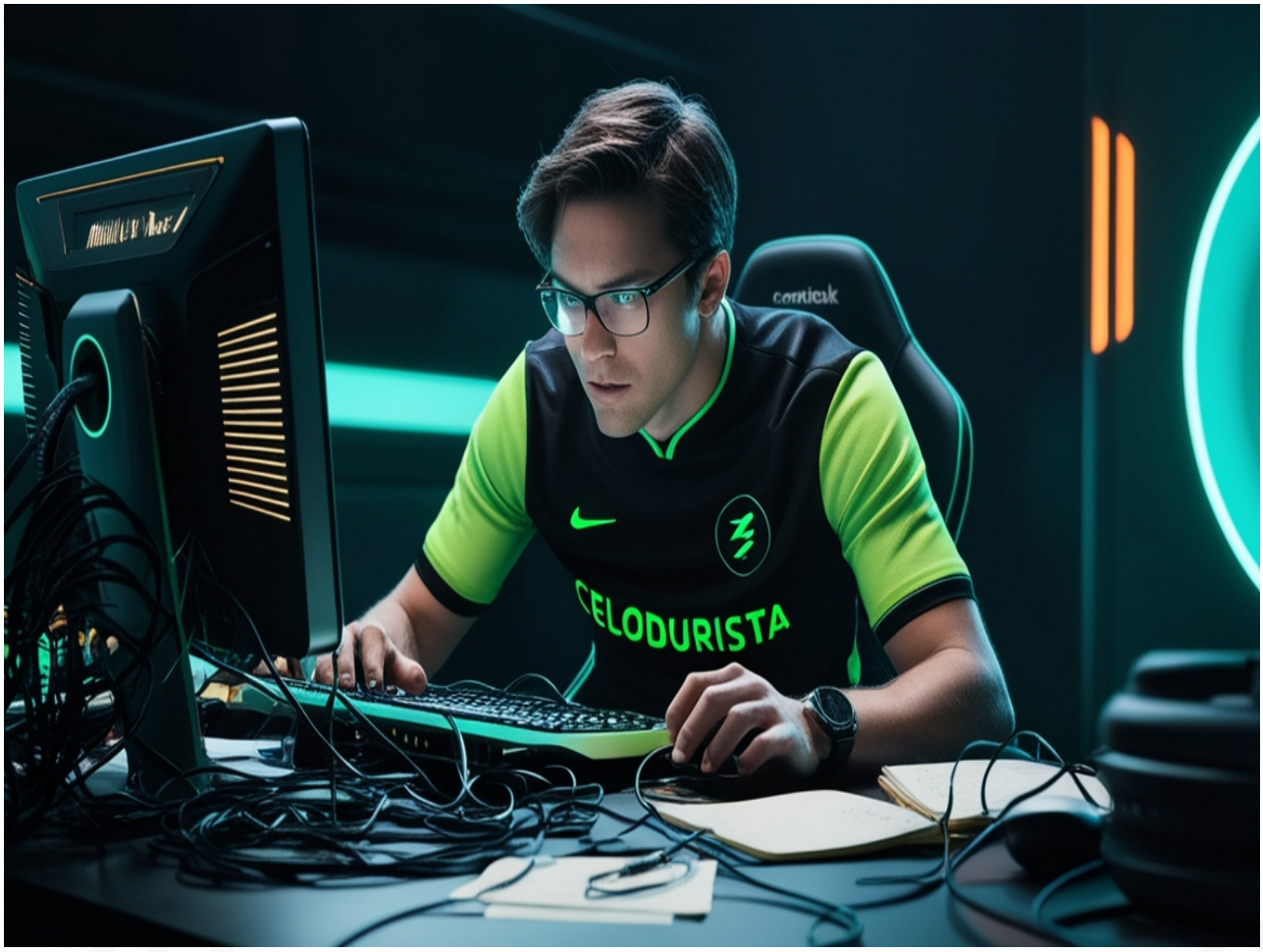
A developer's professional growth is not measured only by the number of languages mastered or frameworks explored, but by their ability to understand the human, organizational, and economic context in which software lives. This section examines the less “romantic” side of the craft: status dynamics, economic motivations, perceived individual value, and the fragility of professional relationships.

We explore how company culture, labor market forces, structural incentives, and personal bias shape decisions, behaviors, and opportunities. We will talk about recognition (or its absence), identity inflation behind keyboards, functional mercenarism, the economic leverage of compensation, and organizational shortsightedness in retaining or losing talent.

The goal is awareness: understanding the mechanisms that determine how we are evaluated, why certain profiles are retained (or discarded), and how to maintain professional integrity while navigating expectations, ego structures, asymmetries of information, and paradoxes of modern engineering culture.

This dimension requires not only technical ability, but lucidity, pragmatic ethics, and a systemic view of one’s trajectory. Mastering it means transforming your work from mere code delivery into the intentional construction of a durable, antifragile professional identity.

# The "Hardcore" Attitude of Programmers



In the world of programming, there's a curious phenomenon we might call "hardcore-ism." This term describes a stubborn resistance to the evolution of software development tools and practices.

Once upon a time, when resources were scarce and precious, programmers had to be incredibly ingenious to squeeze every drop of power from their systems. This necessity forged habits and mindsets that, in some cases, have survived well beyond their practical utility, transforming into a sort of professional folklore.

```
640K ought to be enough for anybody
```

This famous phrase attributed to Bill Gates, although he has denied saying it, aptly represents the attitude of many "hardcore" programmers towards technological innovation. For these programmers, the idea of using modern tools like advanced IDEs, graphical debuggers, or AI assistants seems almost heretical, a violation of the fundamental principles of "real" programming, a symptom of weakness and incompetence.

## Historical Roots

In the early days of computing, when computers were powered by punch cards, programming meant working with systems that had limited memory and processors. Programmers of that era had to be true artists of optimization, capable of writing code that was both functional and extremely efficient.

This era produced masterpieces of ingenuity and practices that still survive in resource-constrained environments: have you ever tried programming for microcontrollers or embedded systems? Here, every byte of memory and every clock cycle counts, and the art of optimization is still very much alive, even if some cracks are starting to show on the horizon.

Over time, as hardware evolved, many of these limitations have disappeared, but the ethic of "doing more with less" has remained deeply rooted in programming

culture, sometimes transforming into an attitude of resistance towards tools and practices that could simplify and speed up development work.

```
"The most efficient code is the code you don't have to write"
```

## The Manifesto of Hardcore-ism

To understand what kind of programmer you are, here's a little test to evaluate your degree of "purity." Try to gauge how much you agree or disagree with these statements:

### 1. The IDE is useless

A programmer should be spartan: the fewer tools they use, the more competent they are.

```
"To write code, all I need is Notepad (or Vi) and the CLI"
```

This statement is often made with a mix of pride and nostalgia. Those who support it seem to want to communicate: "I am a real programmer; I don't need any help." This position deliberately ignores the advantages offered by modern IDEs.

Personally, I've seen programmers use commands like:

```
copy con: pippo.prg
```

with the same pride as someone getting top marks in school.

However, let's not underestimate the advantages of modern IDEs and first of all, let's get rid of the idea that IDEs are just fancy text editors. They are powerful tools that integrate numerous features to improve productivity and code quality:

- Debugging: Breakpoints, real-time variable inspection and alteration,

step-by-step code execution, call stack, and much more.

• Refactoring: Renaming variables or functions throughout the project with a single click, extracting methods, or safely reorganizing code.

• Static analysis: Identifying potential bugs, style violations, or problematic patterns before execution.

• Integration with version control systems: Managing branches, commits, and merges directly from the IDE interface.

• Support for frameworks and libraries: Autocompletion not just for base libraries but also specific suggestions for the frameworks used in the project.

These are just some of the reasons why, when I enter an editor invented in the '70s, I feel like I've lost an arm.

To be intellectually honest, we can't deny that in some situations, an IDE might be overkill, such as when making a quick change or working on remote systems with limited resources. In these cases, minimal editors or anything that opens a text console might be the best choice. For everyday work and complex projects, categorically refusing to use a modern IDE means depriving oneself of tools that can significantly improve code quality and programmer productivity, as well as the entire team's (yes, because the code is not just yours, but belongs to everyone working on it).

**2. The CLI is the solution to all problems**

Let's not kid ourselves; the CLI has undeniable charm. Years of movies with hackers hunched over mechanical keyboards and black screens with green text have shaped generations of programmers. What could be more powerful than controlling a computer by typing text commands?

This approach, although it has the allure of a high school crush, has a series of limitations that are often overlooked:

• Learning curve: Memorizing a few commands is easy; memorizing dozens or hundreds of commands with all their options can be daunting for newcomers and those who don't use the CLI daily.

• Data visualization: Some information is simply easier to understand when presented graphically.

• Complex operations: Certain activities, like managing branches in a Git repository, can become much more intuitive with a visual representation.

Git is a great example of how CLI and GUI can coexist and complement each other. While Git's CLI offers granular control and the ability to automate operations through scripts, GUI tools like VSCode integrations can make complex operations more accessible, such as:

• Viewing commit history with a branch graph

• Performing interactive rebases

• Resolving merge conflicts with visual diff tools

As always, the truth lies in the middle, and the wisest approach is to master both tools. Use the CLI for quick operations and to create automated scripts, but don't hesitate to switch to a GUI when it can offer a clearer view or a more efficient workflow.

**3. We don't use Windows because programmers use Linux (or MacOS)**

I can't count the nights I've spent reading and commenting on the "religious wars" of operating systems.

```
"Winzoz" doesn't work
Linux is great with its 200 versions, all incompatible
Forget it, with MacOS everything works
Look, Apple charges you twice what that computer is worth
```

One could go on forever, creating flame wars full of hate and ignorance, but the reality is that every operating system has its pros and cons, and it's wrong to claim that one is absolutely superior to the others.

A programmer should rise above these barroom chats. This doesn't limit their freedom to feel comfortable on one system over another, but it's absolutely counterproductive to put on blinders and ignore the existence of other operating systems besides the preferred one.

Let's consider the benefits of working daily on different platforms:

• Cross-platform: Developing and testing on multiple platforms ensures that the software works correctly for a wide user base. "Write once, run anywhere" is an important goal for many applications, and even if languages ensure this can be true, every programmer does their part to make sure it isn't.

• Flexibility: Many companies use mixed environments, and the ability to adapt is a competitive advantage.

• Understanding differences: Working on different systems helps better understand the peculiarities of each, improving debugging and optimization skills.

Instead of dogmatically sticking to a single operating system, a more productive approach is to choose the right tool for the right job, maintaining the flexibility to move between different platforms when necessary.

**4. The debugger is for the weak**

Let's debunk a myth:

```
A "real programmer" writes perfect code on the first try
```

This story has been circulating in the programming world for years. More or less all famous programmers have crafted this image for themselves, and each of us

is ready to tell it at the end of the evening when the beers are gone.

The reality is that debugging and testing are essential and unavoidable parts of the software development process, and this mentality of resisting the use of debugging tools, seeing them as a sort of "crutch," is something that needs to be overcome.

I can't count the times I've taken over software considered "finished," full of working tests, and gradually problems emerged, uncovered use cases, analysis and design issues: no, the illusion that the first draft of a program can generate perfect software and that tests are enough to understand what works or doesn't is just an illusion.

In these situations: logs, debugging, and new tests become necessary to understand what doesn't work and how to fix it.

**5. The code is all in my head**

Analysis, knowledge sharing, and documentation are often overlooked aspects of programming.

```
I have it all in my head
```

There is no less productive phrase than this, which stifles any kind of discussion and collaboration.

There is a romantic narrative of the programmer as a solitary genius, capable of holding entire complex systems in their mind and seeing their software as Neo saw the Matrix.

This approach has a fundamental flaw: the human mind, though wonderful, has limits in the amount of information it can retain.

Not all of us are Dennis Nedry, the programmer from Jurassic Park who knew all

the park's code by heart, and even if we were, let's remember what happened to him.

Forgetting the past is a form of protection activated by our brain to avoid going crazy. This is why it's important to document code, share knowledge, and work in teams.

But even if one's mind had infinite capacity, there are other reasons why "code in my head" is a problem:

• Collaboration: If the code's functioning is clear only in the mind of the person who wrote it, it becomes difficult for others to contribute or maintain the project.

• Prone to errors: Without clear documentation or comments in the code, it's easy to forget important details or make incorrect assumptions.

• Complicated onboarding: New team members will have difficulty understanding and contributing to the project.

Therefore, the "hardcore" programmer needs to realize that code is a collaborative product, and not sharing information is not the best approach to keep one's job, but the best way to lose it.

Let's encourage these programmers to document, write Wikis, do code reviews, use diagrams and schematics to share knowledge.

An approach that values documentation and knowledge sharing not only makes the project more robust and maintainable but also contributes to the professional growth of the entire team.

**6. Programmers don't use ChatGPT**

Let's face it: artificial intelligences are anything but intelligent.

```
Look at how many mistakes ChatGPT makes; it can't replace a programmer
```

Hallucinations, but especially the incorrect and superficial use of AIs, lead many programmers to think they are unusable tools.

No, AIs are not search engines; they are linguistic aggregation tools that can learn our work context, and they should be used as such.

Overlaying this theory are the increasingly prevalent theories that AIs will replace programmers in the future.

The "hardcore" programmer doesn't use these tools because they don't need them, because they don't work, because "I am better."

All true, but only in part. The reality is that AIs are tools that can help programmers write code faster and with fewer errors, but it doesn't take an hour to achieve this result; it takes weeks of use to understand how to best use this tool and quickly identify its strengths and weaknesses.

AI should be seen as an enhancement tool, like the evolution of modern IDE autocompletion, but capable of extending completion to a broader and more complex context.

I recently watched the movie Atlas, not for Jennifer Lopez, as many of you might think, but to enrich my mind with new images of possible futures. In this film, AI is seen as a complement to human work, and both improve and enhance each other, working in symbiosis.

For those who spent their childhood afternoons watching Star Trek: it's the evolution of the Vulcan mind meld or the Trill symbiont.

There are many aspects of a programmer's life that benefit from AI:

• Boilerplate generation: AI can quickly produce basic code structures, allowing programmers to focus on more complex and creative aspects.

• Assisted debugging: Models like ChatGPT can help identify errors in code and suggest possible solutions.

• Exploring new technologies: AI can provide explanations and usage examples for frameworks or libraries the programmer is unfamiliar with.

• Optimization: Suggestions for improving code efficiency or readability.

• Testing: Generating automatic tests to verify the correct functioning of one's work.

Instead of categorically rejecting the use of AIs, programmers should consider them as tools that can improve their productivity and the quality of their work.

## Conclusion

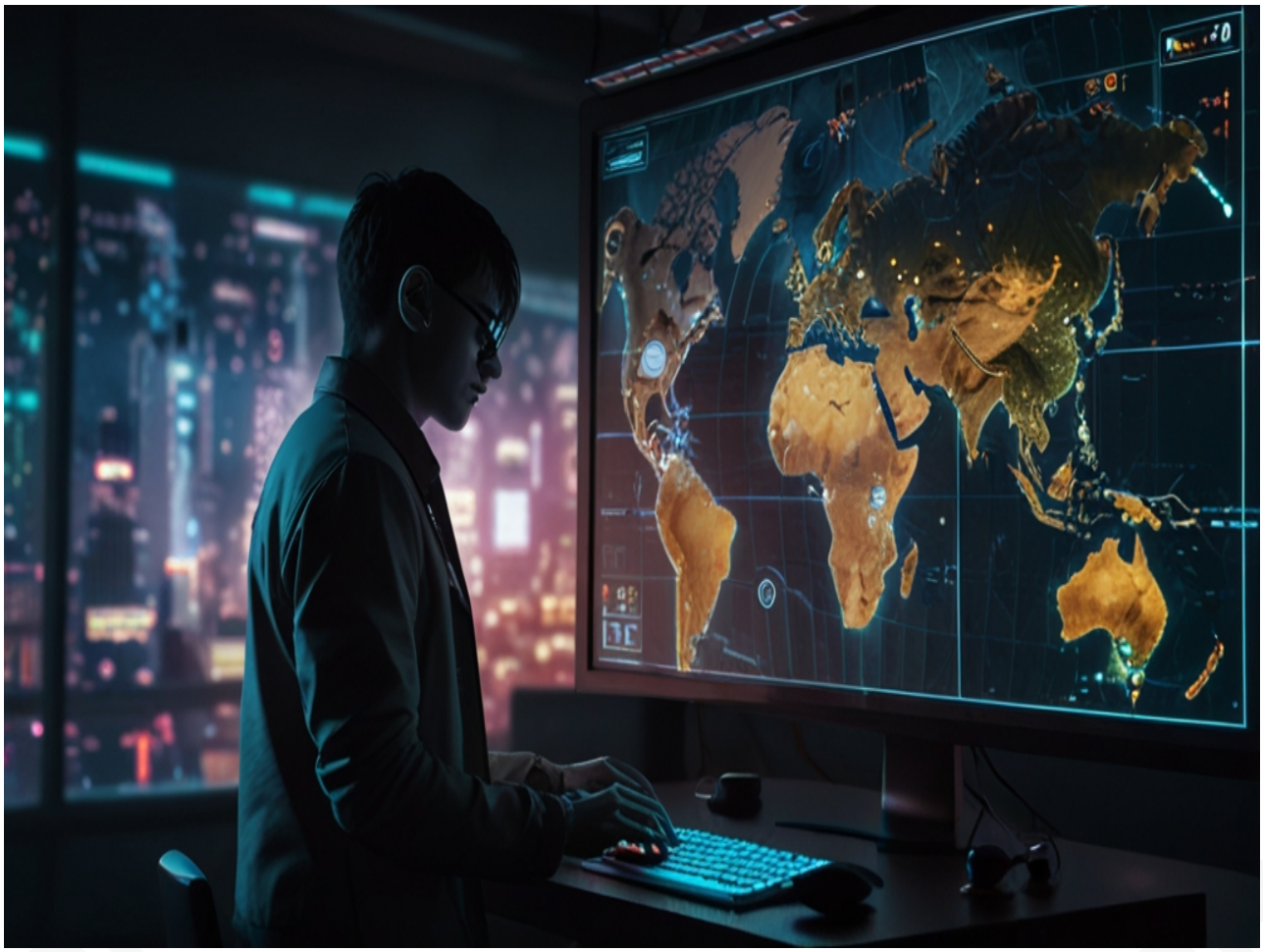
"Hardcore-ism" in programming, though rooted in history as a symbol of ingenuity and optimization, risks becoming a hindrance to innovation and efficiency in the modern software development world.

A more balanced approach recognizes the value of tradition and experience but remains open to new technologies and methodologies that can improve the development process.

The true mark of an experienced programmer is not dogmatic adherence to past practices but the ability to critically evaluate and adopt the tools and practices best suited to each specific situation.

I admire the hardcore programmers for their tenacity, but I'm sure that if they had more courage, they could reap great benefits from reevaluating their positions.

# Programmers as the New Mercenaries: The Evolution of Work in the IT Sector



Once upon a time, building a product with a team of programmers was synonymous with long-term stability. Changes in personnel were rare, limited to events like retirements, personal relocations, burnout, and occasionally, irresistible job offers. But the winds of change have blown fiercely: the spread of the Internet and the recent pandemic have accelerated a process that seemed inevitable: the advent of remote work.

Today, even someone living in a secluded farmhouse in the Alps can collaborate with a company in a distant metropolis as easily as they could work for a local business. The IT work landscape has transformed, and with it, the rules of the game.

## From Centralized to Decentralized: A New Paradigm

We've shifted from a model of strong localization of companies and personnel to a revolutionary concept of distributed work. Today, talking about "full remote" companies is no longer heresy but a well-established reality. Employees work from home or various locations, often in different countries. This transformation has opened the doors to an increasing number of professionals, including those less inclined to change, who now find themselves inundated with a continuous stream of job offers, unthinkable just a few years ago.

In the digital age, even lists of companies offering remote work have become a reality. In Italy, for example, the GitHub project "Awesome Italia Remote" collects Italian companies that offer full remote jobs, complete with required technologies and application pages:

```
https://github.com/italiaremote/awesome-italia-remote
```

This metamorphosis of work reflects what economist Richard Baldwin has called "the great convergence." In his book "The Great Convergence: Information Technology and the New Globalization," Baldwin states: "The impact of information technology is creating a new wave of globalization that allows

services to be provided remotely, radically changing the global work landscape."

This new reality is a fertile ground of opportunities and challenges, both for workers and companies. For IT professionals passionate about their work, free from strong corporate ties, and seeking higher salaries or stimulating work environments, this could be considered a golden age. Flexibility and opportunities have grown exponentially, opening previously unimaginable horizons.

## From the '80s to Today: A Journey Through IT Work

I fondly remember my first job at a software company. The team was a close-knit group of people who had worked together for years, all living just a few kilometers apart. In that context, stability seemed a given, and external job offers were rare and often unappealing.

That corporate "paradise," where no effort was needed to retain employees, is now a faded memory.

Today, companies face a titanic challenge: retaining their talent. Employee retention has become a crucial corporate mission. Replacing a team member is not only costly but also requires a significant investment of precious time for training and integration.

In the current landscape, a programmer is no longer just a "code monkey." They are a multifaceted professional who embodies a constellation of skills that go far beyond mere coding: product knowledge, understanding of business dynamics, empathy with the end users of the software they develop.

Modern software requires professionals with increasingly specialized skills, both technological and product-related. And it is especially the latter that require time to be honed and perfected.

As labor economist David Autor from MIT astutely observes: "Companies are

increasingly investing in firm-specific human capital, making workers more productive in their current roles."

Specialization is a double-edged sword: on one hand, it represents invaluable corporate value; on the other, it can become a risk for employee mobility. The concentration of skills in a few brilliant minds can make the company vulnerable to the loss of fundamental knowledge in case of resignations or retirements.

To tackle this titanic challenge, many companies are implementing innovative retention strategies worthy of a science fiction novel. These include training expense reimbursement programs, enticing stock option offers, regular salary reviews, internal mobility opportunities, and short-term assignments that resemble space missions. Additionally, they are investing in continuous training, both for technical skills and soft skills, offering greater flexibility in work modes and the chance to experiment with new technologies as if they were explorers in uncharted lands.

A few years ago, all these opportunities were reserved for a select few, high-level professionals, or long-term managers. Today, they have become the norm, a must-have for any company that wants to remain competitive in an increasingly globalized and competitive job market.

In the IT realm, training is the lifeblood of developers. Finding companies that invest in training, both internally and by offering budgets for individual growth, is like discovering an oasis in the desert. It represents an invaluable added value for professional and personal growth, a reason to stay anchored to the company instead of being tempted by sirens that do not guarantee equal growth opportunities.

But beware: the IT job market is not a uniform monolith. There are abysmal geographical disparities: in some areas, training is a mirage, and companies prefer to outsource skills rather than cultivate them in-house. Competition has become a global arena, where local companies find themselves competing not

only among themselves but also against tech giants that seem straight out of a sci-fi movie.

## The Disparity Between Large and Small Companies: A Digital Abyss

When talking about disparity, the mind immediately goes to geographical differences, dictated by the territory where people work. But in the IT world, where geography is losing significance, the real disparity is in what companies can offer. And the offerings from large companies are increasingly enticing compared to those from smaller entities.

This disparity creates a wage gap that, within a geographical area, can be mitigated, but on a global level becomes a canyon. This wage abyss is a sword of Damocles for small companies desperately trying to retain the best talents.

We can thus talk about a true "digital migration": people remain physically in their territory, but their minds and skills travel through the network, working for companies located anywhere in the world. This digital brain drain represents a titanic challenge for companies trying to retain the best talents: creating reasons to keep people becomes a Herculean task.

## How to Tackle These Challenges? Strategies for a New World of Work

Properly addressing the challenge of knowledge management is a fundamental step for any company that wants to survive in this digital jungle. It is crucial to avoid concentrating essential skills in a single person, as if they were an irreplaceable oracle. Instead, aim to create autonomous and self-sufficient projects, like ecosystems capable of thriving independently. The loss of an expert technician can be a devastating blow, leading to a significant reduction in corporate capital, not only in terms of technical skills but also in knowledge of processes and corporate culture.

From the professionals' perspective, it is important to consider that changing jobs

solely for economic reasons may not always be the wisest choice, especially for young people at the beginning of their professional odyssey. Moving through different positions and situations can provide valuable experience and invaluable maturity, but it is essential to find a balance. Money is important, of course, but it is not the only treasure to seek when evaluating a job opportunity.

During my early years of work, I remember a phrase that struck me like a bolt from the blue:

```
not less than 2, not more than 5
```

I later heard this mantra in a thousand different contexts, but that time it particularly shook me. The phrase referred to the number of years a professional should spend at a company before changing jobs. Less than two years could be interpreted as a lack of stability, more than five as a lack of ambition. This concept, which may seem like a fossil in a rapidly evolving world, still holds some validity, especially in a sector like IT, where the speed of change is comparable to that of light.

There is also another often underestimated aspect that is important to learn when deciding to embark on a programmer's life: any project tackled, in the first months of development, seems like the Garden of Eden. Problems? Rare. And even when they arise, they can be overcome with relative ease, like jumping over a puddle.

But after the first few months, projects start to grow like young titans, becoming increasingly large and complex. The demands, both from clients and management, turn into mountains to climb.

Over time, providing consistent and functional responses to requests becomes a challenge worthy of Sisyphus. It is at this moment that true programmers emerge, those capable of making a product work stably, taking into account a myriad of aspects that never surface during analysis and early versions.

Reaching that level means becoming true professionals, capable of tackling any challenge and solving any problem. Changing projects every six months can increase horizontal knowledge, but at the expense of vertical knowledge, which often makes the difference between creating a software product and making it work.

## Conclusions: Navigating the New World of IT Work

In conclusion, we have entered an era where teams are as fluid as water, and stability has become a concept of the past, a museum relic. Companies must adapt to this new reality, offering competitive conditions and a stimulating work environment to attract and retain talent. At the same time, professionals must carefully evaluate opportunities, considering not only the economic aspect but also professional and personal growth, like explorers in search of the Holy Grail.

Technology is changing the nature of work faster than many organizations can adapt, like a high-speed train racing while stations desperately try to keep up.

At the same time, programmers must be careful not to be deceived by easy corporate relocations and easy money, as they risk giving the impression of being mere code mercenaries rather than passionate and competent individuals in their work. The true value of a programmer is not measured only in lines of code or salary but in the ability to create, innovate, and leave a lasting mark in the digital world.

In this new technological Wild West, only those who can balance ambition and loyalty, technical skills and soft skills, will emerge as true pioneers of coding. The future belongs to those who can navigate these tumultuous waters with wisdom, adaptability, and an unwavering passion for their craft.

# A Better Salary Isn't Enough to Motivate Change

Periodically, I take time to reflect on what I've accomplished, to consider whether my behavior has been appropriate, and to think about areas where I could improve. Everyone makes mistakes, underestimates, or overestimates situations; revisiting them with a cool, critical mind helps uncover answers that often don't surface in the heat of the moment.

We ourselves change over time, and with us, our priorities, goals, and values evolve.

Today's analysis is dedicated to a series of events that occurred at the end of 2023, which made me reflect on how difficult it is to get people to change, even when presented with an enticing offer—or at least, that's how I saw it.

From time to time, I need to integrate new members into my development team, either to replace those who have left the company or to expand the team for new projects.

Leaving a company is a normal practice that, as I've described before, every manager should anticipate, though it is often underestimated.

To quote Jane Austen:

```
No one is ever too old to change, because change is the only
constant in life.
```

Therefore, we shouldn't be alarmed if a team member decides to leave, but we must be ready to manage the change, sometimes abruptly.

There are many reasons for "leaving." In my case, it involved programmers who left the company for other opportunities, retirements (yes, it happens, it's not a mirage), and relocations: it's wrong to force a programmer to stay on a project they don't feel connected to. Either we motivate them, or it's better to move them to another project with different challenges, to avoid harming both them and, by

extension, the team they're part of.

This time, I needed to find someone to replace a programmer who was retiring: standard procedure, managed well in advance to avoid disruptions and ensure a smooth handover.

## Great, but What's the Problem?

The problem arose during the selection phase when new programmers declined my offer. And no, the reason wasn't the salary, which was higher and aligned with the candidates' expectations.

In the past, I thought salary was the main reason to entice a programmer to change companies, influenced by the countless times I'd read it was the first thing people considered when deciding to change jobs, but I had to rethink that.

I focused too much on salary, a victim of the fact that it's one of the few objective parameters we can evaluate and because it's the most discussed and first evaluated parameter in many programmer forums during any change.

Let's take a step back and look at the needs I had to meet.

The search was for an experienced Java programmer interested in joining our development team for a backend product used across Europe.

Focusing the request immediately is crucial when approaching a candidate for the first time: it saves time for both parties and avoids unnecessary time-wasting.

Many programmers hate working on projects they don't know, don't understand, or don't find stimulating. By discussing the project and the visibility it offers right away, you convey to the candidate how important and visible their work will be.

The second aspect I address is technology.

Working with technologies considered "obsolete" or "old" is a deterrent for many programmers.

```
"I don't want to work on Java, it's an old language, I'm not interested."
```

is a phrase I've heard many times. Just as I've often heard:

```
"What version of Java do you use? 8? No thanks, I'm not interested."
```

Even if the language or technologies used aren't an issue, being able to use a recent version of the language or technologies they don't know is an incentive for many programmers. Similarly, making it clear that the project is continuously evolving and that efforts are being made to bring the product to a more recent version of the language is an additional incentive, as long as it's not unfulfilled.

In my case, it was a project that started in Java 11 with a completed port to Java 17, using various libraries and frameworks.

The important point is that I wanted to convey that the project was continuously evolving and that the candidate's work would be crucial to the project's success.

The third aspect I address is salary.

Knowing it's an important aspect, I mention the salary I can offer during the first interview to avoid creating false expectations: it's an advantage for everyone. The candidate already knows what they can expect, and I know if the candidate is interested or not.

By immediately focusing on the project, goals, and compensation, the candidate

already has a complete picture of the situation and can decide whether to continue the selection process.

Partly due to the clarity, partly due to demonstrating that the tech stack was up-to-date, and partly due to disclosing the salary upfront, I received an adequate number of applications and, after a selection phase, identified some promising candidates. They were suitable for me, the team, the managers, and, most importantly, all the candidate's requests were met: working in a familiar field, using recent technologies, having a higher salary.

Despite the encouraging premises that led me to believe everything could be resolved positively, I received unexpected negative responses:

```
"I'm not ready for it."
```

## When Salary Isn't Enough

The main reason some candidates rejected our offer was that they didn't feel ready for the change: their current job, though less stimulating, less interesting, and less paid, was still a secure and familiar job.

Change, even for the better, was perceived as a risk, a leap into the unknown.

This made me realize that even if you're in a company that doesn't satisfy you and you have every reason to change, if you're not inclined to change, you won't, or at least not for a minimal change. Stronger motivations or a dialogue that makes the change feel not like a moment of stress but a moment of growth and enhancement are needed.

I've noticed this phenomenon especially among more mature individuals, who prefer to stay in familiar contexts rather than face change, even if beneficial.

The "comfort zone," even if problematic, can make us accept toxic situations as

normal.

```
The greatest obstacle to change is not ignorance or resistance, but the illusion that what we are doing now is safe.
```

This also made me think of all the times someone confessed to me that they were dissatisfied with their job but didn't want to change for fear of not finding something better.

```
The fear of change is a common feeling, but it can be overcome with the right strategies.
```

Based on what I've observed, some strategies that can help overcome the fear of change are:

• Small Steps: for those who dislike "brutal" change, the technique of small steps can help. Do you feel unsatisfied with what you're doing? Start studying what you want to do. Dedicate time to it every day, and over time, you'll have enough knowledge to make the leap you've always wanted.

• Growth Mindset: don't view challenges with fear, but as opportunities to grow and evolve your situation. Having a growth-oriented mindset allows you to almost seamlessly break free from your mental cage.

• Visualization: don't view change negatively, but try to take your mind beyond and think about what positive things could happen once you've changed.

• Social Support: seeking support from colleagues, friends, or mentors helps better understand what can happen and provides the encouragement needed to overcome the mental obstacle preventing change.

**The Mistakes I Made**

It's easy to blame the candidates if someone decides not to accept your offer. Addressing the issue intelligently should instead prompt the question: where did I go wrong?

If we can't understand what didn't work, we won't be able to improve.

In my case, I tried to find the answer beyond salary, beyond the project, beyond the technologies used.

```
And if behind that "I'm not ready for it" there was actually an
"it's not exactly what I'm looking for"?
```

I tried to understand what might have driven a programmer not to accept an offer that seemed enticing.

One of the reasons I considered was certainly the corporate environment in which the candidate would have to work, or at least the way I portrayed it during the interview.

Some talk about "corporate appeal," which is the attractiveness a company has towards a candidate.

Corporate appeal is a combination of factors ranging from work style to corporate culture, from benefits offered to growth opportunities, from training possibilities to working in a stimulating environment.

Corporate appeal is an important factor that is often underestimated, but it can make the difference between a candidate accepting the offer and one who rejects it.

Showing that your company has values, believes in valuing people, offers growth and training opportunities, and provides a stimulating and comfortable work environment can make the difference between a candidate accepting the offer

and one who rejects it.

## Conclusions

Thinking that just offering a higher salary can motivate change is a mistake I've made throughout my life.

I haven't brought on board people who could have made a difference for my team and my project.

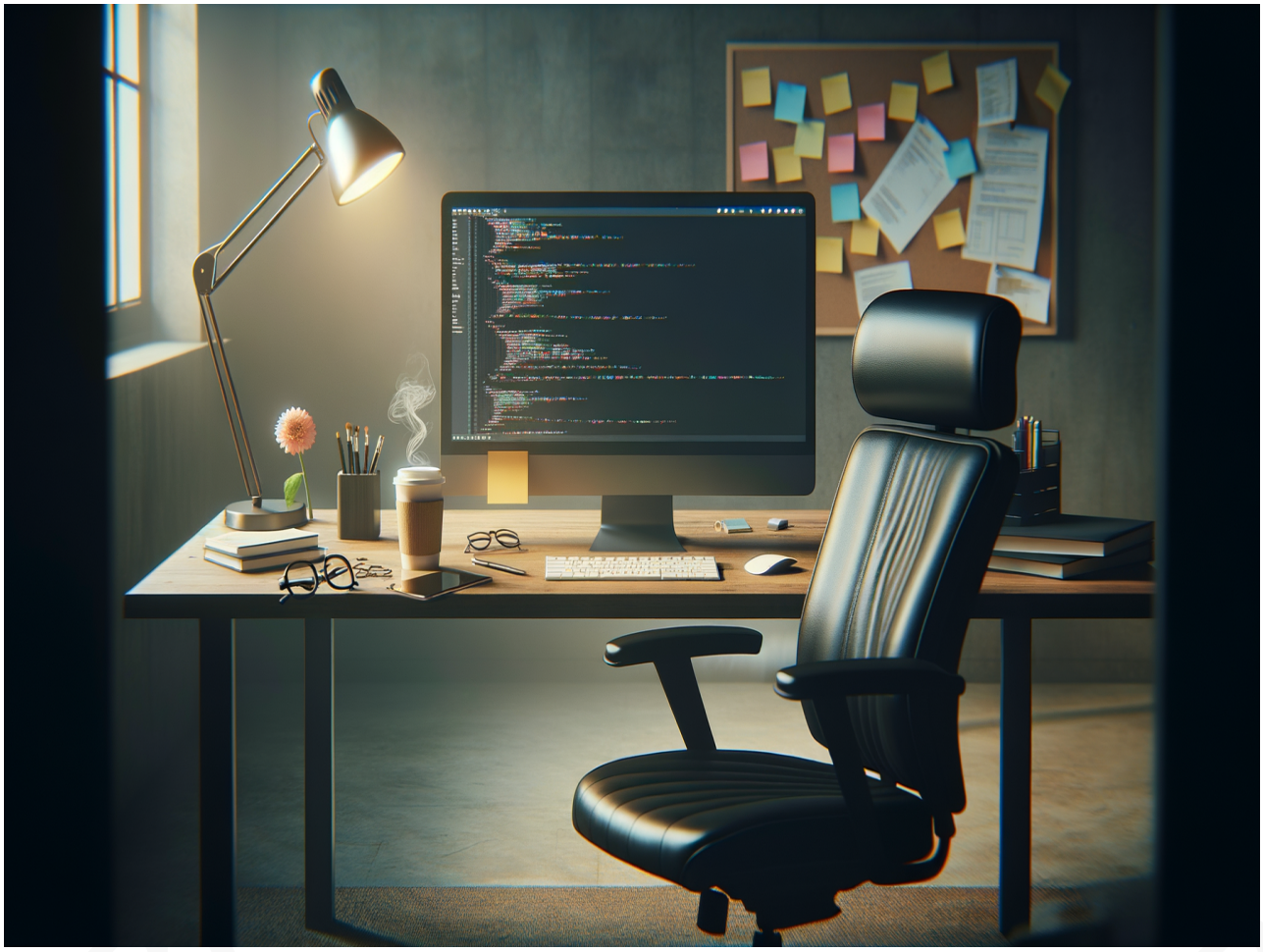
This taught me that the main job every manager should do is to make their company attractive to candidates, to make them perceive that working there is an opportunity for growth and enhancement.

Making the candidate feel that they will be able to work independently, with the right tools, that they will grow year by year, and be an important piece for the company's success.

At that point, many aspects that were previously considered fundamental become secondary.

Salary, the project, and the technologies used become just details, secondary aspects compared to corporate appeal and the growth opportunities the company offers.

# Losing a Programmer Means Losing a Treasure



An old saying goes, "A friend is a treasure." The positive meaning of this phrase lies in the fact that true friendship—the kind that lets you sleep peacefully because you know that no matter what happens, you'll always have a friend to rely on, who helps you out of desire, not obligation—is rare.

Similarly, a good programmer is a treasure for a company. With their technical skills, experience, and problem-solving abilities, they can make the difference between a project's success and failure.

However, if we look at the flip side, "Losing a programmer means losing a treasure." Too often, the loss of someone who has accumulated experience and skills within the company over time and now decides to move elsewhere is underestimated.

I've seen too many companies suffer setbacks because someone left, whether it was a programmer, a project manager, or an analyst. This article aims to reflect on what it means to lose a programmer and how companies can prevent it from happening.

## The Professional Life of a Programmer

Programmers, like all professionals, go through several significant phases during their careers. However, it's often possible to identify a recurring pattern that appears frequently in this field.

The first phase is joining a project. This is an initial period, of varying length, where the person is introduced to the work environment. It's a time characterized by intense study alternating with practical work, often under the guidance of a more experienced programmer.

Next comes the start of actual work activities. In this phase, the programmer begins working on the first assigned tasks, getting familiar with the project and the team members. They are not yet fully productive but start understanding the

project's dynamics and the workgroup.

The third phase is professional maturity. The programmer begins to work independently, knows who to interact with, understands the company's processes deeply, identifies problems, and develops effective solutions. It's a time of great growth and professional satisfaction.

Finally, there is the settling phase, where the programmer has almost all the project's code in hand. The excitement and curiosity that characterized the initial phases gradually diminish. It shifts from a situation where "everything is possible," filled with euphoria and curiosity, to a phase of greater awareness, sometimes even thinking that "everything has already been done" or "nothing more can be done."

It's precisely at this moment that the programmer starts questioning their professional future, their current job, the project, and the company.

As Aristotle says in "Nicomachean Ethics":

```
Happiness is not a moment or a state that is achieved once and
for all, but a process that lasts a lifetime and requires
constant commitment to the exercise of virtue.
```

For this reason, companies should proactively intervene to support their programmers during this phase.

## The Loss of Motivation

This phase represents one of the most critical moments in a work life: the gradual loss of motivation, ambitions, prospects, and goals. Everything seems gray or crystallized in a seemingly unchangeable situation.

Some professionals face this phase constructively, channeling their energies into

improving existing code, better structuring the project, optimizing processes, and perfecting documentation. They dedicate themselves to those backlog activities that are often postponed and find space only in moments of apparent stasis.

Others, however, begin to develop a deep sense of discouragement, boredom, and frustration. They start looking beyond, seeking new stimuli, new projects, new company realities. And from discouragement to jumping to a new company, the step is surprisingly short.

This process, when viewed closely, is not entirely negative. In many professions, one reaches a saturation point that pushes for change. The unique advantage for programmers is that, if they have the right skills, they can make this change relatively easily compared to other professions. They can transform their work situation quickly and often without even leaving their workstation.

## The Loss of a Programmer

Contrary to what one might superficially think, a programmer is not simply a replaceable element in a company's value chain. This is especially true in contexts where the work team is not very large, and the skills are highly specific, held by a few key people.

Not everyone works in large, well-structured, and redundant companies. Often, one finds themselves in small companies where the development team is under ten people, and losing a programmer can have devastating consequences.

To further complicate the situation, there are projects themselves that lack adequate documentation, are the result of years of overlapping work, and where processes are not optimally structured. In these situations, holistic knowledge, where present, remains concentrated in the hands of a limited number of people.

In this scenario, losing a programmer becomes a serious problem because you truly lose a treasure and, in some cases, even the effective control of the project

they were following.

Even in more structured corporate environments, where teams are well-organized and the code is carefully documented and tested, there is a tendency to look for someone who can replicate exactly the skills and activities of the departing programmer.

This aspiration often proves unrealistic because each programmer has their distinctive style, established habits, and specific skills. It's not easy to find someone who can replace them perfectly.

In these situations, you often hear: "I've been looking for someone for months, but I can't find them." This seemingly banal statement actually contains several significant truths: it highlights how the person to be replaced had specific skills that the current team cannot cover and suggests that the compensation they received was probably undervalued compared to the real value they brought to the organization.

Another frequently underestimated aspect in these circumstances is the expectation that the new programmer can fully or partially cover the predecessor's work from day one. Surprisingly, some still harbor this illusion.

Instead, it's crucial to account for the fact that, for a significant period, the team will not be able to maintain the same performance levels. This issue becomes even more critical if the departing programmer is one of the most experienced or if the team is composed of a limited number of people.

## How to Prevent Losing a Programmer

Awareness of a programmer's value should drive companies to create a work environment that encourages their retention and professional development. The key element in this regard is ensuring an adequate level of autonomy in daily work. A programmer should have the freedom to organize their time and

activities, choosing the most effective ways to achieve the set goals. This autonomy does not mean isolation or lack of coordination with the team but rather the ability to best express their creativity and technical skills without excessive procedural constraints.

The company must also ensure that the programmer has all the necessary tools to perform their job optimally. This means not only providing adequate hardware and software but also ensuring access to training resources, documentation, and technical support. Too often, companies underestimate the negative impact that obsolete or inadequate tools can have on a programmer's motivation and productivity. The frustration of having to struggle daily with technical limitations can be a strong incentive to seek opportunities elsewhere.

Plato, in "The Republic, Book IV," said, speaking of potters:

```
If due to poverty he cannot procure the tools or other utensils
essential to his art,
he will produce inferior products and make inferior craftsmen
of his children or others to whom he teaches his craft.
```

A crucial aspect, often underestimated, is making the programmer truly feel part of the project they are working on. This goes beyond mere technical involvement: it means involving them in strategic decisions, listening to their opinions on architectural choices, and involving them in planning future activities. A programmer who feels truly part of the project develops a sense of belonging and responsibility that goes beyond the simple professional relationship. This emotional bond with the project and the team can be a powerful retention factor, especially in difficult times or when alternative opportunities arise.

When a programmer feels they have a say in decisions regarding their work, can rely on tools that meet the challenges they face, and are an integral part of a meaningful project, the likelihood of them seeking professional alternatives decreases significantly. It's not just about retaining a qualified professional but creating conditions where they can best express their potential, thus contributing

not only to the project's success but also to the growth of the entire team.

## Conclusions

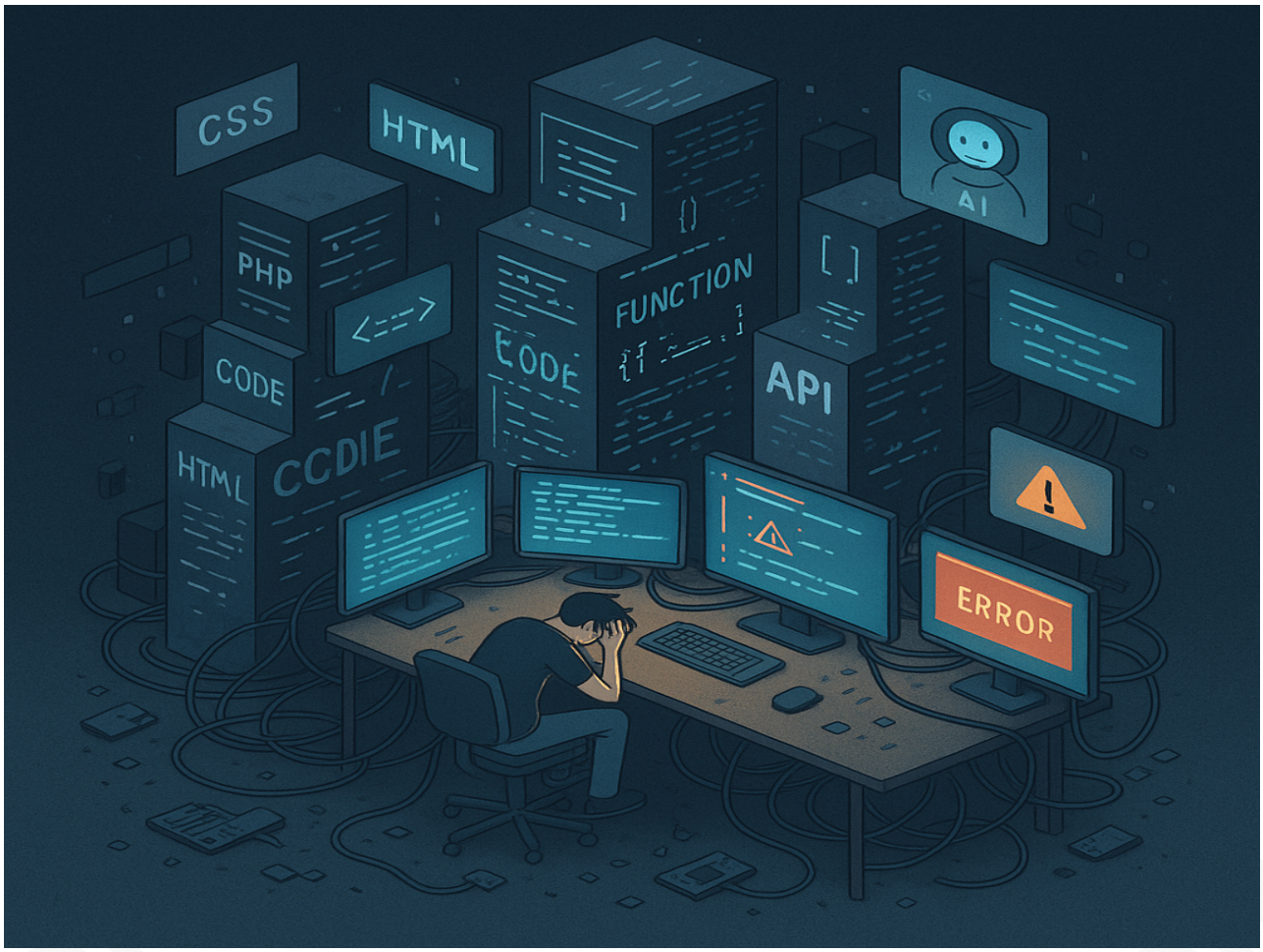
The professional journey of a programmer is a complex one, characterized by various phases of growth, moments of enthusiasm, and periods of reflection. It's important for companies to be aware of these critical moments and actively commit to creating a work environment that encourages the retention and development of their programmers.

The treasure metaphor, with which we opened this reflection, gains even more significance when considering the wealth of knowledge, experience, and value a programmer brings. It's not just about technical skills but a wealth of deep understanding of processes, project dynamics, and team relationships.

When a company loses a programmer, it doesn't just lose a resource but a piece of its history and future.

The challenge for modern organizations is not only to attract talent but to create an environment where they can grow, express themselves, and find continuous stimulation. Only through a concrete commitment to valuing their programmers, providing them with autonomy, adequate tools, and a real sense of belonging, can companies hope to retain this treasure. In a world where technology evolves rapidly and skills are increasingly valuable, investing in the satisfaction and growth of their programmers is no longer just a choice but a strategic necessity for long-term success.

**What if the weak link in programming is the programmer themself?**



While artificial intelligence promises to revolutionize software development by accelerating every phase of the process, a disturbing paradox emerges: the programmer themself may have become the main obstacle to the speed the market demands. In an ecosystem where automation generates code at unimaginable rates, human supervision introduces latencies, checks, and slowdowns that contradict the primary goal of efficiency.

## A bit of history

Let's try to understand the world of programming by reflecting on the normal evolution of a project and thinking about how a programmer moves over time.

When we take our first steps in this world, we begin to study the first languages and learn about the first components, libraries, and frameworks that can help us in our work.

The study of best practices and an increasingly deep knowledge of the tools used tend to focus our work on a known path that we tend to replicate, in part or in whole, in every new project.

If we reflect on programming, it is often a linear process: today we create a functional data entry, tomorrow we implement efficient controls, then we think about data abstraction mechanisms, create APIs, optimize performance, increase security measures, and so on.

Each project becomes the sum of previous experiences, appropriately reorganized: perhaps it's better to design the APIs from the start, security would be better implemented as a first step, let's use a consolidated control library, this authentication service, and so on. However, the sum of experiences, in a perspective of continuous evolution, can hide an insidious trap.

## The growing weight of complexity

The real problem lies not in the code itself, but in the complexity that inexorably

accumulates over time. Each new project represents a starting point that is rarely truly new: the programmer immediately inserts everything they have learned in previous years, unconsciously accumulating layers of- complexity.

Initially, this complexity appears manageable, but it progressively turns into an increasingly difficult burden to bear. Each new feature, each bug to be fixed, each client request adds another layer that settles on the existing code, creating a stratification that requires increasingly specialized skills to be managed.

This dynamic clashes head-on with the expectations of the current market, where speed has become the dominant parameter. Customers want everything and they want it now:

```
Create a website like Apple's, but in a week since you have
everything ready.
The prototype must be ready by tonight.
The bug must be fixed by today.
```

and to finish with a quote that I must have heard an infinite number of times:

```
When do you want this feature?
For yesterday!
```

The time pressure, always present, becomes increasingly constant and inescapable.

## The illusion of time compression

Faced with this pressure, the industry has tried various automation strategies. In the past, there were CASE (Computer-Aided Software Engineering) tools, code factories specialized for specific categories of problems. With the evolution of needs, these tools have transformed into modern frameworks: more flexible and general-purpose compared to the verticality of CASE, but still complex and not always intuitive to use.

The arrival of artificial intelligence has represented the ultimate promise: programs that write code for us, that solve problems automatically, that save time and effort. However, this apparent solution clashes with an inescapable reality: the programmer remains the controller of this process: a human controller, with all their limitations and reaction times.

Even by delegating the writing of the code to the AI, irreducible times persist: learning, adaptation, debugging, testing, and human supervision. As long as this supervision remains necessary, the overall time cannot be drastically reduced. It is precisely in this dynamic that the programmer emerges as a potential weak link: the human element that introduces overhead and latencies into the automation process.

## But how are AIs helping us and evolving?

2025 was the year of "vibe coding": programming assisted by artificial intelligence where each problem is solved through iterative prompts. If something doesn't work, a new prompt is generated; if something needs to be modified, yet another prompt.

This evolution has transformed AI from a simple support tool to the main work tool.

I have been working in "Vibe" mode for months, first with prompts entered in the various LLMs, then through Copilot, then with CLI tools like Gemini and Copilot Agent, up to more immersive tools.

After banging your head against it for a while, you start to see its limits and inconsistencies, although you often think they are more yours than the machine's you are using.

Like me, other prominent figures, such as Jack Dorsey, have begun to use AI for

their "weekend projects," creating applications like BitChat entirely through automatically generated code.

However, direct experience with these tools reveals a complex dynamic. Relying completely on the "vibe" to generate large portions of code produces a high number of elements to be re-checked and fixed. The interpretative ability of the AI, although impressive, can easily deviate from the intended direction.

The most effective approach has proven to be that of fragmentation: using the AI for small specific tasks, checking and testing each component, and then assembling them manually. This method allows for quality control while still taking advantage of the acceleration offered by automation. It is analogous to the process of domestication: it is necessary to know, understand, and guide the tool before it can be used effectively.

## When the programmer stops: a near future scenario

The- growing acceleration of development processes raises a critical question: what happens when the programmer is not available? In the past, the absence of a developer simply meant a proportional slowdown. Release cycles were annual, the pressure for new features limited.

In the present, acceleration has transformed every interruption into a potential systemic problem. In the near future (2-3 years), it is plausible that projects will continue to evolve in an unsupervised manner during the programmer's absence, with the AI autonomously generating code, resolving issues, and managing branches.

Upon their return, the programmer would be faced with dozens of branches to examine, issues to understand, bugs to analyze to verify the correctness of the automatically generated code. The mental load of this situation is significant: a sense of guilt for the absence, anxiety from missed supervision, pressure to regain control of a process that has continued to evolve autonomously.

This scenario can easily degenerate into a vicious cycle of burnout, leaving the company, or loss of professional interest, summarized in the dangerous attitude:

```
"Let's accept this pull request, the AI has already done all
the work, what could go wrong?"
```

## The future of the programmer: beyond the immediate present

Projecting ourselves a few years into the future, the evolution of the programmer's role appears inevitable. AI already surpasses any typist in speed, is beginning to produce code of higher quality than that of many developers, and can make mistakes at an unprecedented speed. However, it remains pure power without strategic thinking.

The transformation of the programmer's role will likely follow this trajectory: less time dedicated to writing code directly, more time interacting with AIs and in business-oriented work. This change will require awareness and adequate preparation, as it will completely overturn the current conception of programming, pushing professionals beyond their current skills.

In this new paradigm, a person will describe a problem in a few sentences and a language model specialized in analysis will generate the prompts for a model specialized in writing the code. The programmer will evolve from a manual coder to a guide, tester, and, perhaps, a refiner of the code generated by the AI.

This transformation will entail the disappearance of some current practices: traditional debugging, the need to meticulously document the code, the importance of variables with explicit names. In parallel, new programming languages designed for interpretation by machines rather than for human readability will likely emerge.

## The human mind: the insurmountable limit

In this scenario of continuous acceleration, driven efficiency, and frenzied automation, there is one element that cannot be compressed: the human mind. Shaped by millions of years of evolution, our mind has developed mechanisms to solve complex problems through slow and thoughtful processes.

Depriving this "biological machine" of the time needed to process information and overloading it with continuous inputs produces only one result: a functional block. In a world where every slowdown generates disastrous domino effects, taking sick days or rest days risks becoming an inaccessible luxury.

In inadequately structured organizations, where each person is individually responsible for specific activities without adequate distribution of workloads, the mental and physical burden that each programmer must bear will become progressively unsustainable.

## Conclusions: the weak link that can break the chain

Programming is undergoing an epochal transformation with the advent of AI. We must learn to live with these technologies while preserving our humanity and critical thinking skills. The speed required by the market is not always sustainable, and sometimes it is preferable to slow down to ensure the quality of the work.

However, this choice exposes us to the risk of being perceived as the weak link in the production chain, the element that prevents time compression and the achievement of immediate results. Faced with this prospect, we will have to question the necessity and usefulness of our role, or whether we have become a burden to the organization.

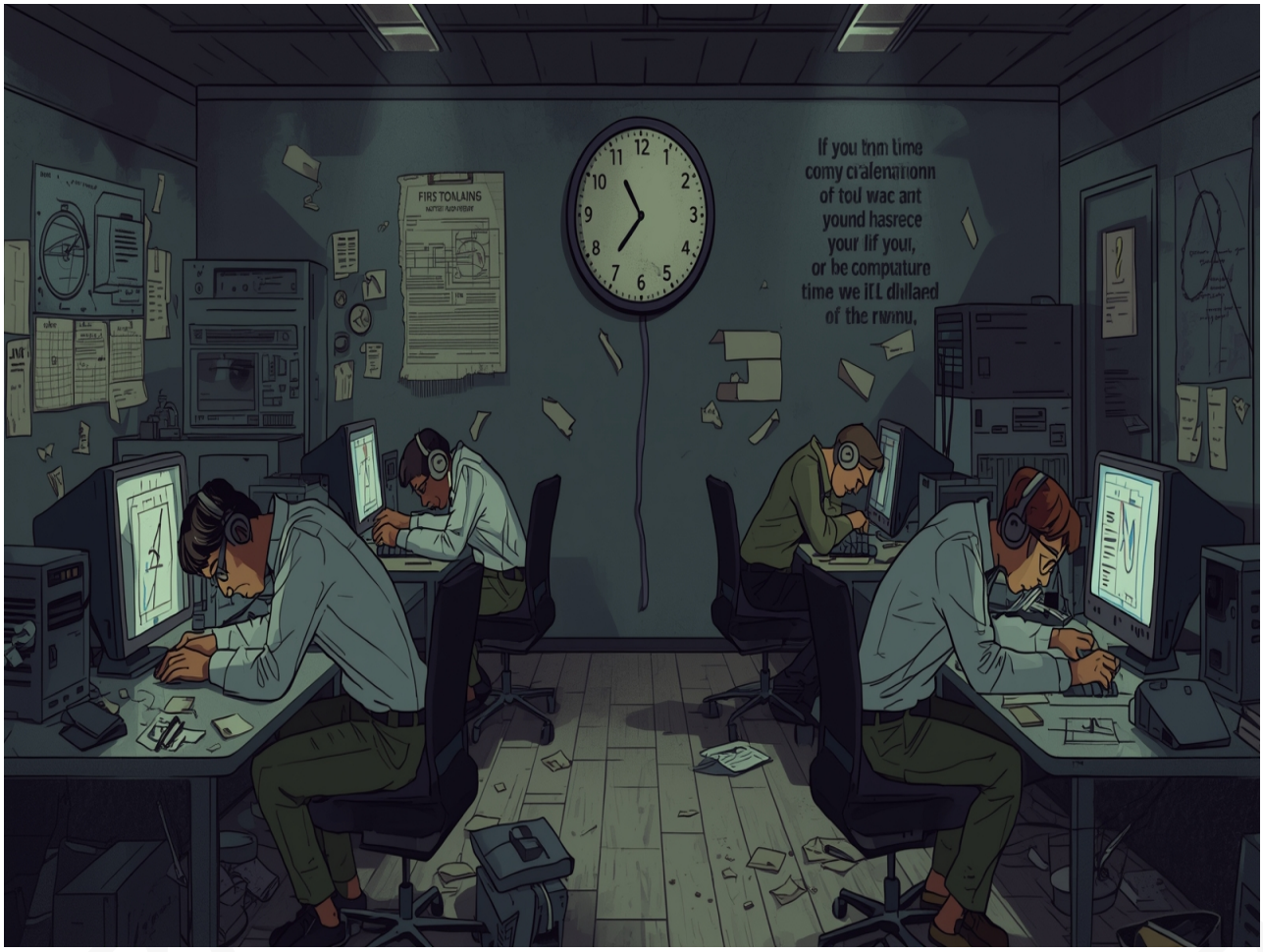
I conclude with a reflection by Stanisław Jerzy Lec that I consider particularly significant:

```
"The weakest link in the chain is also the strongest because it can break it."
```

The programmer, while being considered the weak link in the automation process, also holds the key to breaking the chains of complexity and speed imposed by the system.

Our strength lies in our ability to adapt, learn, and evolve. Although the future may label us as the weak link in the chain, our task is to transform this apparent weakness into the strong point of our work and our organizations, otherwise we will soon become obsolete and replaceable by increasingly intelligent and autonomous machines.

# Why Tech Startups Are Doomed to Die



During the scorching August days, I found myself chatting with a childhood friend.

We both work in IT: me at technical tables, where people get angry over a misplaced bit, and him at commercial tables, where they try to sell ideas that a mass of nerds spawn endlessly.

Different backgrounds, parallel ways of seeing life and work, but with many similarities when it comes to failures and successes.

The reflection we started over a cappuccino and croissant quickly transformed into an analysis of the dynamics that lead to the failure of tech startups.

Guido, I hope you'll forgive me if I borrow some of your insights for writing this article.

## Welcome to the Slaughterhouse of Good Intentions

According to some studies, 90% of startups are destined to fail.

explodingtopics

Saying that 90% fail is a polite way to describe a graveyard of ambitions, a massacre of good technological intentions: "I'll start dieting in September" is the parallel of "I'll start a startup that will change the world in September."

The truth, the one nobody dares to say out loud, is that most of these "startups" were never real companies. They were lab experiments, expensive tech hobbies disguised as businesses, led by coding geniuses who had no clue how to build a business.

```
Our startup won a hackathon
```

The most experienced investors now exclaim "who gives a damn" to these kinds of

statements, since these "awards" mask the most common, most predictable, and least discussed cause of death: the suicidal absence of commercial acumen. Having a spitting contest at a tech event doesn't mean having a sustainable business model.

A startup founded and led exclusively by techies isn't simply ***destined*** to fail; it's already dead from the start. Its failure isn't a question of ***if***, but of ***when*** the investors' money or the founders' patience will run out. You need strategy, you need a commercial soul, you need a real and not fictitious need—aspects that most of the time aren't remotely in a techie's wheelhouse, who thinks more about their code's speed than listening to what the market is screaming at them.

In a recent interview, Enrico Pandian recalled the 2-week strategy: "If you can't validate your idea in 2 weeks, you're probably building a castle in the sand."

interview

## Anatomy of a Predicted Failure: The Deadly Plague of "Tech-Only" Startups

CB Insights

When these analyses are done, politically correct jargon is used, attempting to obscure the true nature of the problem. Here we don't have these constraints and can assert without filters that it's not the market that's absent: it's the founder who didn't bother to look for it. Funds don't "run out" by magic: they're burned by disastrous strategic decisions. If until yesterday you spent your day lining up bits, you can't wake up and magically become the wizard of business strategy.

## Being Irrelevant (No Market Need): Creating Solutions Nobody Needs

This fierce evidence manifested brutally in the blockchain world: hundreds of solutions that provided guarantees nobody felt the need for, except the people

who had built the startup.

Ideas that arrived too early or too late? We'll never know, but what we do know is that these are ideas that interest few people and not the vast audience they were targeting.

In the AI world we're in the same situation: an infinite number of products, now too similar. This time though we have the advantage that the average person has the clear feeling that these are tools capable of increasing their value, and where there's perception of value there can be market.

The absence of market need is the number one cause of death: an impressive percentage of startups, whose numbers hover around 40%, is the tangible sign that you can have brilliant ideas, which in most cases are brilliant only inside our computer and not in people's heads.

This isn't an accident; it's premeditated murder of one's own idea, perpetrated by founders in love with their technological solution but completely uninterested in the problem it should solve.

This phenomenon, known as "tech-solutionism," is the occupational disease of technical teams. They are trained to solve complex puzzles and create elegant architectures; they are not trained to leave the office to verify if anyone, in the real world, cares about their creation.

## Why do startups fail? A core competency deficit model

The results are staggering: the two skills whose absence was most closely correlated with failure were "information-seeking" and "customer service orientation."

If you're programmers, software engineers, or technicians, you might have an idea of what these skills are, but you certainly haven't dedicated much time to

refining them.

Seeking information implies actively validating market hypotheses, while customer orientation means prioritizing user needs over technological purity: whether a software flow is optimized matters to a customer as much as knowing if your kitchen color is red or yellow. If instead the "delivery address" field is missing (or doesn't work): yes, this is indeed a major flaw in your software solution.

A team composed exclusively of techies is, by its very nature, structurally deficient in these vital areas. Therefore, "No Market Need" isn't a market failure; it's a direct and unequivocal failure of the team to perform the most basic commercial functions.

A homogeneous team is destined to build brilliant but perfectly useless products, because nobody within it has the task, competence, or incentive to ask the simplest and most important question of all: "Who needs this?"

Silicon Valley

## Running Out of Cash but Being Full of Code

Running out of funds is the second most cited cause of death. We're talking about estimates around 30-35%.

Running out of money is an elegant way to say that startups weren't able to generate new money.

In 1999 Jeff Bezos was interviewed and asked: "Your company is worth billions, but every time I look at the balance sheet I see it's losing money: your company has never made a profit" and Jeff serenely responds "That's true, but we're investing to grow."

If the company invoices and grows every year, losing money isn't a

problem—you'll always find an investor. If the pennies entrusted to you don't produce any kind of result, the problem is deeper and you're destined to drain all your funds and close or be absorbed by someone who can get your business started, but your hyper-technical mind isn't capable of doing it.

The invisible killer of startups is a mathematical equation completely unknown to techies:

```
CAC > CTLV
```

The customer acquisition cost (Customer Acquisition Cost) is higher than the value that customer will generate over time (Customer Transactional Lifetime Value).

A team of only techies can be obsessed with product metrics like server uptime, latency, or loading speed, but completely ignore the business metrics that determine the company's life or death.

The inability to manage cash flow is a direct consequence of this financial illiteracy. Technical founders, driven by the desire to create the "perfect" product, continue to invest resources in development, hiring more engineers and adding features, without a corresponding strategy to generate revenue.

In my distant past, I remember a conversation with a French company that produced a library for Clipper whose name now escapes me. They had landed a huge contract that allowed the company a positive explosion: hiring, expanding their offering, and profits.

How was this contract born? The sales department had sensed a great opportunity in selling software to a client. That software didn't exist when it was requested: in one afternoon they drew some product screenshots with "PAINT," saying it existed but needed 6 months to configure it at the client's site, given the project's complexity.

Good: the client was convinced, signed the contract, and in 6 months they moved the entire project team to realize that application that didn't exist, but which the client was convinced was ready.

Now imagine a purely technical approach to the same problem:

```
we don't have the application, but we can make it in 6 months
```

The technical result would be identical, but this phrase, in a client's mind, leaves the feeling that you might not succeed; saying you already have it but need to customize it generates a completely different perception of control.

Along the same lines, there's an interview circulating online with Claudio Cecchetto who, after listening to the cassette of "Hanno ucciso l'uomo ragno," asked 883: "There are only 5 tracks, do you have 2 more?" and they said: "yes, they're in the studio, let's go get them and bring them."

In reality they didn't exist: they locked themselves in a room for two days and came out with two more songs.

Saying "we don't have anything, but we can make them in 2 days" would have conveyed uncertainty and perhaps wouldn't have created the opportunity they managed to seize.

But a techie would never have said something like that: to say it required someone who, deep down, was more of an entrepreneur than a technician.

What is genius? As Perozzi said:

```
Fantasy, intuition, decision and speed of execution
```

## The Plague of Invisibility (Poor Marketing)

If from some points of view the invisibility cloak is a significant advantage, Harry Potter knows this well from the countless times he wore it, but if you have a company: having a product that nobody knows about isn't something to boast about.

14% of startups fail from this dark evil. If you follow the mantra: "if you build it, they will come," you're destined to present your product only in the living rooms of other nerds like you.

Technical founders often harbor deep contempt for marketing, considering it "fluff," dishonest manipulation, or, at best, a secondary activity to do "later" when the product is perfect. No: this is a death sentence. The perfect product doesn't exist. Think of many tools you started using: the first versions were extremely limited, yet you hooked onto them immediately, following them at stalking level.

According to Peter Thiel, co-founder of PayPal:

```
The number one cause of company failure is poor distribution,
not the product
```

Don't be afraid to make your product known, to ally with those who have a distribution chain, a sales force, powerful marketing: they're the ones who will give you the possibility to expand, not your product.

The inability to do marketing isn't a simple skill gap: it's a cultural prejudice, an ideological blind spot. In failure analysis, a recurring theme emerges: the inability to do marketing was a function of founders who liked programming or building products but didn't like the idea of promoting them.

## Wrong Team (Not the Right Team)

Having the "wrong team" is one of the main causes of failure, responsible for closing almost a quarter of startups.

Having a wrong team doesn't mean a team of incompetent people, but a team that's too homogeneous: imagine a soccer match with a team made up of only forwards: all phenomenal, but unable to counter the opposing team. Defeat becomes an inevitable element.

The real problem, almost always, is the lack of a co-founder with commercial DNA: if you're a team of only programmers, one of you must throw away the keyboard, or better yet you need to find a NON-programmer, someone who can't write a "Hello World" (not even with ChatGPT) but can sell refrigerators to Eskimos.

One of the friend companies I've seen grow the most is composed of a brilliant technical mind and a commercial person with great market vision, whose union created a winning balance.

## Case Study: Stripe – The Programmers Who Learned to Sell

At first glance, Stripe seems like the exception that proves the rule. Founded by two Irish brothers, Patrick and John Collison, both programming prodigies, it's a company built by techies for other techies (developers).

But if we don't stop at this crude analysis, we discover that their success doesn't contradict our thesis, but reinforces it even more powerfully.

The Collison brothers didn't succeed ***despite*** being techies; they succeeded because they combined their technical genius with exceptional commercial and strategic ability.

They understood they had to actively sell it. Their initial go-to-market strategy, now legendary and known as the "Collison Installation," is a masterpiece of founder-led sales.

When they met a potential customer, they didn't give them a PowerPoint presentation. They asked for their laptop, and in a few minutes, before their eyes,

they integrated Stripe into their product.

Instead of ***talking*** about value, they ***demonstrated*** it unequivocally.

## Stop Writing Code, Start Selling

Being a good programmer isn't enough, knowing how to produce solutions isn't enough—you need a marketing soul, a sales soul that doesn't have its roots in a programming language, but in different experiences and studies.

The greatest salesperson I've known was a summer village entertainer: he had great empathy and likability. He made you talk, understood your needs, and knew how to persuade you.

If you're a group of programmers, look for someone like that, who brutally breaks out of your patterns, who knows how to enhance your products and talk to people. Look for someone with a sales network who knows how to channel what you're building. At that point the road becomes downhill and above all you'll have a broader team, capable of better facing market challenges.

A programmer doesn't need a clone, they need a partner. You need to talk with potential customers and understand what they're looking for. You don't need to write code if we don't know which direction to go: better to lose days talking than days writing code we'll then throw away because it doesn't satisfy any requirements.

The world is full of elegant code. What's scarce are companies capable of transforming that code into real value for their customers.

Stop hiding behind your text editor. Get out and sell. It's the only way to not end up in the graveyard of geniuses.

# Biography

Matteo Baccan is a professional software engineer and trainer with over 30 years of experience in the IT industry.

He has worked for several companies and organizations, dealing with design, development, testing, and management of web and desktop applications, using various languages and technologies. He is also a passionate computer science educator, author of numerous articles, books, and online courses aimed at all levels of expertise.

He runs a website and a YouTube channel where he shares video tutorials, interviews, reviews, and programming tips.

Active in open-source communities, he regularly participates in programming events and competitions.

He defines himself as a "realistic dreamer" who loves to experiment, innovate, and share his knowledge and passions, following the motto: "Never stop learning, because life never stops teaching."

# Indice