Agile software development is a set of ideas about how a team of professionals can work on a product while at the same time improving the way they work with each other. This is where the word agile comes from. Agility means the capability of changing to better fit the given circumstances. As such, Agile lacks carved-in-stone definitions. It has guidelines and a very vibrant community of people eager to discuss advantages, disadvantages, deployment difficulties and side effects of given Agile practices. But in a way, it is possible to define Agile in negative terms, that is by saying what it isn't. Agile arrived in prominence in the early 2000s as an alternative approach to work management fundamentally from then popular methods that came to software development from other industries and relied heavily on preparation and documentation.

Imagine that you're building a bridge.

How do you do it? You can't just build anything and let people use it. If the bridge crumbles, people will die. We have to know that the bridge is safe before even the first person walks on it. What's more, it's not easy to modify a bridge. You can't just build it, test it, the fight that, no, no, it's not good enough, take it down and build it again. It would be way too expensive.

The development process used to build bridges has to, therefore, focus on defining exact requirements on the design and testing on small scale models. When all this is done, the construction team steps in and builds the bridge exactly to the documentation they get. They are not supposed to have their own ideas and alter anything, or they'll risk the future of the whole enterprise. To some extent, the same goes for the car industry, housing, ships and airplanes. All those areas where construction and testing are expensive, so it makes sense to have everything planned out and without possible mistakes before the workers actually start to build anything. People who came to software development from industries that work in this way brought their management practices with them. Of course, they saw that software development is different, but the practices they were used to have already grown into and mixed with the whole corporate culture and it was difficult to modify them.

Personally, I blame IBM.

For a long time IBM specialized in big data processing machines, predecessors of current supercomputers. They were mainly interested in expensive hardware and their software department grew naturally as an offspring of it. They already had people on the management positions taking care of talking to customers, defining requirements and writing documentation. It was easy for them to see programmers as construction workers who just have to build what others already agreed on. IBM's success meant that the other especially big corporations like IBM itself, decided to use their methodology as a model for their own. But in software development it is easy to test in a full scale or at least something close to the full scale environment, just as it's relatively easy to modify an already working product.

On the other hand, **the level of complexity in software is such that, during gathering requirements and planning, it's difficult to see the whole picture**. Even a careful designer may forget about a detail, which will make it impossible to implement the whole project the way it was if the errors are then caught during the implementation or the testing phase, someone has to go to the designers, tell them they screwed up, and then everyone has a major problem. Even worse, if the product goes to the customer and then comes back with a list of things which don't work as they should or are totally not what the customer asked for. Agile points out that software development is fundamentally different from other industries. So different that we should reorganize the whole process around developers, that is designers, programmers, of the development process itself. What if it doesn't have to be written down in a book the size of a brick? What if it can be a result of interaction between developers, managers and agile coaches, that is, people interested in how all of this works and how to make it better?

In broad terms, Agile can be summarized in the following four principles:

1. **Individuals and interactions over processes and tools.** Written rules are downplayed in Agile in favor of asking everyone involved how the current workflow and the set of rules suit them, and if not, then what they think should be changed. The feedback is then turned into lists of more or less clearly defined issues, and proposals on how to solve them are being collected and eventually some of them are implemented, so that in this particular case, in this company, the process is improved. Later on, an ambitious Agile coach may write down what was the problem, what were what was done in the end and did it actually help? And she or he can go with all those notes to an agile conference, share them with other agile notes and maybe, just maybe, what was improved in the company A will be also improved in the company B which shares similar characteristics to A. But it's not a general rule. Changes happen on a case by case basis.

2. **Working software over comprehensive documentation.** Importance of documentation is downplayed in favor of working code which should be good enough to be understood by all involved parties. Which I have an issue with because from my experience it's a very idealistic approach. No code is overreadable that documentation is not necessary. But I agree that in a project under development, especially under agile development, the code may change very fast and keeping the documentation up to date can be a burden with no real value.
3. **Customer collaboration over contract negotiation.** Gathering requirements far ahead before their implementation and turning them into into a set of tasks which the developers should stick to, is downplayed in favor of a more direct connection to customers and/or their representatives, so that they can be asked which solution do they prefer from a set of possible options, at a moment closer to when the given solution is going to be implemented. And they can also try out the implemented feature much faster, even before the final product is ready, and leave the feedback for the team so the product might be improved more to the liking of the customer without much drama.
4. **Responding to change over following a plan.** In Agile, the plan of how things should work from the planning board to the working shiny product in the hands of a happy customer, that plan is nothing secret. It can be changed at any moment if needed. The very same rules which we apply to work on the product can be applied to the process itself. Just as the customer may walk in the office in the middle of the development process and say they really wanted something very different, the developers may decide that they are not happy about something in the way they work together and attempts to change it. There is no rule against it. This ability is exactly why we can take Scrum, which is one of the most specific and best documented agile methodologies, poke it with a proverbial stick made of everything that is happening in the company and see what falls off. Sometimes it's retrospectives. I hate retrospectives. If you hate them too, maybe do Scrum but without retrospectives. Sometimes it's the velocity measures. Sure, maybe you don't need to measure velocity. Maybe it takes too much time. If so, do Scrum without measuring velocity. Sometimes it's the Scrum Master.

Beautiful theories often fall apart in practice. Sometimes because they're just wrong, but more often it's simply because of people. People have a history of not fitting well into theories. We are unpredictable, we are not able to separate work from private life, we talk in different languages, sometimes literally. So when you get a bunch of people and make a team out of them and then make them work with other teams also made of people, things happen.

Theories are cherry picked, shortcuts are taken, hacks are developed, mistakes are made, process verb constructions are used to avoid responsibility. But despair not. Fortunately for us, a lot of teams have already tried to use agile in practice for a long time and through trials and errors they somewhat developed a set of practices which more or less work and can be reproduced with confidence, that framework is made of Scrum and Kanban, sometimes sprinkled a bit with extreme programming. Of course, the exact mix differs from one

company to another and also within a company and also changes in time even within one team.

Positions, terms, roles, how they interact and how an ideal team should work together. The theory in Scrum relies a lot on those precise definitions and that people performing the roles should act precisely within the framework defined in Scrum. But the practice is not like that. In practice one person often has to fulfill two or more likely one and a half roles in Scrum. Or maybe a piece of this, a piece of that and a piece of something completely else. Almost always a Scrum team is plunged into a much bigger framework of a large company that doesn't work according to Scrum rules and when they are introduced the rules it, the company, tries to bend them. So, okay, we have a team of developers.

The team has a team leader who can be either a senior developer with a knack for organizing the work of the team or more of a manager type whose main responsibility is organization and communication and coding takes only a part of their time. In theory, the tasks of the team leader should be split among the Scrum Master, the Product Owner, we will get to that, and a senior developer. Scrum doesn't have the role of a team leader. That was one of the things which drew me to it. From a more philosophical and political perspective, Scrum looks quite anarchic in nature. All roles are equally important, and the process relies on that people are professionals and can work out solutions through discussion, without an authoritative figure telling them what to do. The Scrum Master is there to facilitate the discussion and prevent conflicts, not to be in charge.

In practice though, having **a team leader taking charge of the Scrum Master duties is a common sight.** Alright, so we have this person here and team members here. The team members vary in experience and specialize in different areas, but their skills should also overlap partially so that not one developer is indispensable. I talked about it in the video about the real programmer myth. If you let people over-specialize and one of them has to take a break for some reason, you're in trouble. But if you force them all to learn the same things, you probably make them miserable and lose on productivity. One cure of in this case is a QA developer. Someone who should be a very important person in the team, but who is often delegated to a separate QA team and only lend to the dev team. That's okay, things can work like that, especially if the QA team works closely with the dev team. Although, I wouldn't advise the QA developer to work separately, contacting with the others only through emails, slack or jira. That makes the feedback on bugs and overall quality to take way too long. Okay, let's do something. From Scrum comes the idea of a Sprint. And a sprint planning.

Every two or three weeks we sit together for about an hour and plan what our work will be in the next two or three weeks. In theory, a sprint should be a very clearly outlined time of development. With a defined goal which at the end of the sprint is reached and some kind of a product, a new functionality or an evidence of that certain bugs were fixed, is presented to the product owner. We assign ourselves to some tasks, we talk about them on daily standups, we switch if needed, so we pair to do some pair programming. We take new tasks when the old ones are finished and at the end of the sprint we have something to brag about. **Strangely enough, Scrum does not define how exactly do we keep information about the tasks and what's their progress.**

Here's where a task board from **Kanban** comes into play.

A board with post-it notes on it is such a symbolic part of Kanban that people often mistake it for the whole of Kanban. Yet ripping it out of Kanban and sewing it together with is a bit like building up Frankenstein's monster, but on the other hand we do things like this all the time in IT, wiring together and fitting parts of different libraries and frameworks, so it feels strangely natural to do it. Anyway, the board consists of at least three columns: To Do, In Progress and Done. To the left of To Do lies backlog, the bottomless abyss of all the tasks everyone reported at some point in the past. The tasks can be sorted according to their priority, their february, the date of creation and so on. During the planning we decide what can be taken from backlog into to-do. Ideally we should start with an empty board, put some tasks in to-do knowing from experience how much we can do during the sprint, then decide on who does what and slowly during the sprint move the tasks from to-do to in progress and then to done.

Very often between in progress and done there will be also In a review column, a task put in that column should be assigned to another person than the one who worked on it. That reviewer should take a look at the code and/or test it and either put it back in progress together with some notes or move the task to done. At the end of the sprint, all tasks should be in done, the team can congratulate themselves and sit down to plan another sprint. In practice we almost never start with a clean board. There are always tasks left from the previous sprint, either because we were too optimistic about them, or we didn't have full knowledge what they involve. Or they were shots from the fight, tasks introduced to the board in the middle of the sprint. A common solution is to leave the in progress and review columns as they are, clear done and put tasks from to do back to backlog.

Then we discuss if we want to pull them back to to do or maybe something different became more important. But where the tasks come from? One main source of course is bugs. They may be reported by the customers, by the QA team, automated tests or manually testing the app's interface and they might also be found by developers while they work on a different task. The other main force is new or enhanced functionality and here comes the product owner or should come if it was really Scrum.

Product owner is a person in between people who want new functionality and developers. Essentially they're a translator from corporate speak to nerdspeak. Product owner talks to the managers or the sales or whoever needs to be talked to figure out what they want, then talk to the dev team, and as a result of that discussion, the task board gets flooded with new tasks. That role might sometimes be performed by the team leader, but that's a very large burden on top of the usual team leader's work. Another solution, apart from having an actual Scrum-style product owner, is to have feature teams. A feature team is a temporary team consisting of people from different teams gathered together to discuss, plan, implement, test, and finally release new functional It makes sense especially in cross-platform applications where every new feature has its components written in different technologies and you have a team for each of them. For example, you are looking at a cross-platform application right now. Every web application is cross-platform because you have a frontend and a backend part to it. So if we already have two teams, a frontend and a backend team, then it starts to make sense to have even more, like a separate QA team and a team of designers.

Then we pick people from each of these teams and create **a feature team**. Only for the time needed to develop the feature. When the plan is ready and everyone in the feature team more or less knows what to do, the devs selected from our team come back with a handful of new tasks. In Scrum, adding new tasks should happen only at the beginning of the Sprint, but a Sprint is two or three weeks long. It's pretty common that the plan for a new feature is ready somewhere in the middle of it. What do we do then? Well, at the beginning of the Sprint, when we already know that some of the developers will be involved in the new feature, we create a blob task for the whole duration of the sprint and assign it to those people.

Then when they know what does it involve they can break this blob into smaller tasks. Working on a feature will almost always take more than one sprint. It would be awesome if we were able to split the blob task in such a way that new tasks could be put in backlog, pull in to do, done during the sprint and then at the beginning of the new sprint we would have a clear board and we could fill it with more tasks from the blob. In practice though the initial planning of a feature is very Many details become clear only when we actually sit down and implement the damn thing. And quite often they turn out to be new tasks. And they have to be done now, they can't just be put in backlog to wait for the next sprint. So we do that now.

Let's say you start a new project. Or you are in the middle of one, but for some reason things got serious, the project made a jibe, and suddenly you need to hire new people who will save it from an imminent disaster.

Who do you look for? A 10x engineer. That's who. Someone who will almost instantly be productive, who will have no problem working over hours if the compensation is right, and who you can trust with your idea and be sure that he will do whatever is needed to make your dream come true. Then, if the project is successful, you'll not only have the initial version of the product, but also a story. Here is a brainchild of a brilliant software developer. It couldn't happen without him.

And what if the project fails? Well, usually we don't hear much about such projects. A month ago, a thread about such a real program or in this case it was called a 10x engineer, made rounds on Twitter. According to the author, a 10x engineer is someone who keeps all the codebase in his head, prefers to work alone, don't like meetings, don't like mentoring, thinks that teamwork is a waste of time and so on. The author was a manager and he didn't mean to be sarcastic. He appreciated these traits in a programmer. In his opinions, such a person was a cornerstone of a project.

"A company should cherish their 10x engineers and build developer teams around them." The thread was mocked mercilessly, which gave me a bit of hope for the programming community, but the programming Twitter is usually ahead of the rest of the world, and I know from experience that the myth of a real programmer is alive and kicking. Companies would like to be seen as welcoming and inclusive. They say they value teamwork. They're all for personal growth. They want the office to be a fun place to hang out. They want to draw in junior developers who will gain experience and become valuable assets for the company.

But the company's goal is first of all to make money and when it comes to hiring someone, all that progressiveness suddenly has to take a step back. Sure we want a diverse crowd of newcomers, but oh, you know, we're in a very dire position right now and how about this one time we'll look for this very experienced guy who will make sure that the project won't fail. But reliance on brilliant hero developers may hurt the company, especially in the long term. In the short term, yes, it's less risky, and after hiring your hero and putting him on a pedestal, you will probably see a surge in productivity. But if you're not careful, if you let things go this way, one day you may wake up with a number of issues making a negative impact on your project: the quality of the code and the atmosphere in the team. Let's talk about them. Sooner or later, every project moves out of the phase where one person can handle it.

No matter how real your real programmer is. And the company needs more people on the team. So, okay, new people are being hired. **But instead of working on creating a functional team with people of different talents who cooperate and complement each other, some companies choose to build the team around their superstars. In such a scenario, new members become sidekicks.** Their role is to unburden the senior programmer from the more mundane tasks, fixing trivial bugs, writing unit tests, taking care of some particular non-crucial parts of the project, and so on.

Now, it's important to note that the senior developers are not the bad guys here.

They don't have to follow the stereotype. If they are willing to mentor new members and, in time, delegate more and more important tasks to them, the team may still evolve in the good direction, even despite the best company's efforts to sabotage it. Unfortunately, it may happen that your senior developer follows the real programmer stereotype. He focuses on writing code, is unwilling to mentor does not like when during a code review someone tells him that his solution is wrong, he's allergic to the documentation, he, it's almost always a he, assumes that other team members are there to help him, not to learn from him, and definitely not to teach him. And in a way, this system can be stable and effective. Every team member has their position and their tasks, and they all may perform the tasks very well. Junior members either learn on their own or are just happy with the situation.

So if everything goes well, what's there to criticize? Well, let's start with that we have a person in the team with a very crucial role, and a lot of responsibilities nobody else has. Other team members look up to him and try to be like him, or maybe even better than him. It creates a situation which does not promote cooperation but rather competition. **The team is no longer a set of more or less equal colleagues. It becomes a pyramid** with the senior programmer at the top and the rest climbing up and adjusting. Helping each other becomes less important than an individual contribution and that individual contribution is often measured in lines of code.

People who follow the real programmer stereotype better are judged to be more valuable. Those who do not stay behind, get delegated to less important tasks and finally look for another job. But that's not everything. Far from it. A bus factor is a half-serious notion of how many developers have to be hit by a bus before a significant part of the project will be left without anybody having a good idea what's happening in it. In projects where the team is built around a single senior developer, the bus factor is 1. In reality, people rarely get hit by a bus, they just leave the company because someone else made them a better offer, because of a personal conflict, or because they are no longer happy with what they are doing.

Projects built around single senior developers and companies embracing the hero culture are sensitive to the bus factor. Being the most important person in the team is not fun at all. Yes, our rockstar is in charge of the whole project. He gets to decide who does what, he often works individually on the most important parts, other people look up to him, he gets praised when everything goes well, but he also gets blamed when something goes wrong. Oh, even if nobody blames him, he tends to blame himself. Or maybe not, maybe he chooses another unhealthy coping mechanism, and instead of blaming himself, he starts to blame other people. Or maybe it's all good, but the rule of promotion to one's level of incompetence is applied, and the company promotes him to the position of the team leader and starts to treat him more like a manager than a programmer.

So instead of programming now he has to do a lot of things he doesn't like, and he's not good at. Either way, his relationship with the rest of the team, and the project, and the company, tends to get difficult sometimes. And when it does, it may just happen that he will choose to quit. After all, a senior developer will find a good job in no time. Even if the team is a healthy one, this is a difficult situation. If the company embraces the hero culture and the rockstar developer around whom the whole team is built is living, this is a disaster. It means that suddenly large parts of the codebase are no longer maintained by anyone and they are almost always the most complex parts of the code.

There is no good documentation. Real programmers don't like to write documentation after all. The code should be self-evident. If it was hard to write, it should be hard to read and blah blah blah so on. No other team members know it very well. They weren't mentored. Real programmers don't like mentoring.

Maybe they were sometimes doing code reviews of it, but we all know how reviewing code of real programmers by junior team members look like. Ooh, this is weirdly complex and I don't understand it. But Rick surely knows what he's doing and I don't want him to waste more of his time on this review, so how about I'll just click accept. Also, it often means that the code really is weirdly and unnecessarily complex. Programmers working on one piece of code for a longer time tend to stop noticing that their code is unreadable. It's still readable to them after all. But the rest of the team does not know what that code does and how it's different from what it should be doing.

They are not sure which parts are there for a reason and which are redundant. What can be refactored. Where is a bug if there is one. It will take months before they'll build confidence and understanding needed to handle it with ease. And in the meantime, the whole project may go the way of the last season of Game of Thrones.

In my humble opinion, companies tend to look for new employees when it's kind of too late. As a company, we plan a new project, we find funds, we make the plan, we go with it to our developers and oh fuck, we need more developers.

Okay, okay, usually it's not that dire, but you can see the trend. The developers team is often tasked to the roof. There are no free resources, so to say. If somebody leaves or if we get a new project, we need more people. But when a new person joins a team, they usually become productive only after some time. Time, weeks or even months. At start, **having a new person means that now other team members need to schedule some time to teach them**.

So for some time now, not only we still don't have more people, but the people we have have less time than before. The way to solve this is to hire someone already experienced in the technologies we use. Such a person will be much easier to introduce and will probably even know some clever tricks we don't know.

It's a win-win, right? In consequence, job offers are biased towards senior developers. All over LinkedIn, there are offers full of three-letter long acronyms and lists of required skills so big as if this one search candidate was needed to save the whole company from collapsing. Most of those requirements are about coding: programming languages, technologies, years of experience in coding in Difficult, evidence of projects the candidate was involved in, and so on. There's enough of them that candidates tend to think that despite all the sweet talk about inclusivity, personal growth and so on, this is what companies value the most: expert coders at the expense of everything else. So they try to be one. And in a result, they perpetuate the myth.

Okay, so now we have the whole system in place. You have to be very skilled in coding, have years of experience and immense knowledge of technologies in use, because only then you'll find a job where you'll spend your days coding very important stuff and get a lot of money. If you don't have these qualities already, you should spend your time if you can't, well, maybe it's not for you. It's called gatekeeping. It's when a part of the community sets up artificial rules guarding what does it mean to be a real member of the community and what qualities one has to have to become one. In case of programming, these rules focus on great coding skills and superficial traits, which may signal that you have them, like knowledge of certain niche technologies, getting more certificates or having a penis. It's ironic that even though the myth is alive in the community since long, the name Real Programmer started as a parody.

In 1982, Ed Post wrote an essay Real Programmers Don't Eat Quiche based on another parody book popular in these times Real Men Don't Eat Quiche. The book described an exaggerated image of what it means to be a man. The essay did the same about programmers. To be a real programmer was to use punch cards and write in Fortran or assembly languages. The real programmers shunned new fancy languages like Pascal and their fancy features like data structures. The technology changes so fast that even the specific rules of what does it mean to be a real programmer had to change a few times. Nowadays, even a real programmer doesn't use Fortran and don't write directly in the terminal.

IT became so big and diverse that there is no one language or technology which could be recognized as the technology of real programmers. But gatekeeping didn't go away, it just moved to slightly more abstract positions. Now it's more about how advanced your knowledge should be in the technology of your choice, given that it's worthy of being known by a real programmer because of course it can be too easy and how much time you spend learning it and working with it. So what if you don't have time to do it? Or what if you're not particularly interested in any advanced library or framework? What if you have a small child and that's what occupies of your time? What if you're a single mother and you haven't slept in three years anyway, even though you don't maintain an open source product?

What if you have other commitments you can't or don't want to drop? What if you're simply not a workaholic? Or maybe you're just late to the party because for a long time you couldn't afford a good laptop, internet connection and time to learn? Or maybe you decided to switch your career path only in your 30s or 40s or 50s? Well, then you're screwed. At least according to real programmers. Yeah, it seems easy to ignore these voices, and this is exactly what I advise you to do.

But they can be intimidating from time to time. Let's go back to the situation I described a while ago. A team where people climb an imaginary career ladder. They don't even have to compete with each other. They may as well help each other with their tasks. But with the real programmer myth on stage, they still judge themselves according only to its rules. Writing a complex piece of code is a step up.

Failing at it, even if it's a failure only in your mind? Oh boy, you're not a real programmer.

You're a fake. Clearly you shouldn't be here. This is exactly what people told you. This is what you see in the eyes of those who didn't tell you anything. And that guy who said don't worry about it, what he meant was that you'll get one more chance, you'll fail again, and that's it. The impostor syndrome is a well-known psychological condition. It can take many forms and it's better if I make a whole other video about it.

Here I just want to underline that one. It's very common in the programming community. On all levels of experience, from beginners to senior developers, to people who achieved a lot and whose work is globally appreciated. And two, the fact we suffer from it is in big part because of the sneaky thought that maybe we're not good enough, maybe we should have studied more, worked more, maybe we're just stupid and we should leave programming to other, smarter, more committed people. Well, what if we can erase all the IT and start from scratch? What? That's not possible? Damn it.

I don't have all the answers, but I think I know where we can start. We can realize that being a programmer is not only about coding. **Coding is maybe half of the game.** There is a diagram I made for the intro episode and I'm going to get back to it from time to time. Coding is here. Around it are tools we use to build a working application. There's documentation, design and so on. Code reviews.

And then there is teamwork, soft skills, methodologies, time management. And even when it comes to coding, Complex code is not the same as good code. The code is good when it's written in a way that any member of the team can read it, understand it and start working on it in a reasonable amount of time. The Real Programmer myth claims that what is needed is only the small subset of all the skills, namely coding and expert knowledge of advanced tools and libraries. But this subset here is far from being all that is needed. And if we say: okay, so a Real Programmer is someone who knows everything, Well no, I don't think there is one person on the planet who is brilliant at all these things. The set of all skills is just too big.

Each one of us is good at only some of them. And that's good, that's natural. That's why we form teams whose members help each other and complement each other. Yes of course some of the team members are more experienced than others, but on the other hand a junior developer is someone with a fresh look on the fiftam and an ability to to push it in new directions. They can also more easily take over from experienced colleagues, while other senior developers are often already responsible for some parts of the codebase and it's difficult for them to take new responsibilities. It also means that if you are this brilliant guy who's mostly interested in coding with some pretty advanced tools and avoid meetings, there is also a place for you in the team. Maybe just work a little more on your soft skills and a little less on even better coding skills.

Help others, write a documentation, go to meetups and conferences and talk about your work, mentor someone, answer questions on Stack Overflow, that sort of stuff. Please note that the real programmer myth hurts us all, creating unreasonable expectations and a lot of unnecessary stress also for you. It's not an us versus them situation, it's all of us versus the hero culture. Let it go. So, if it was up to me, I would like my company to look more for junior developers, but to do it early enough. And during the interview process we should focus more on their potential, their ability to learn, communicate, willingness to share and test new ideas and things like how they respond when someone spots an error in their work. Sure, technical skills are important.

A junior developer has to already have certain knowledge and experience. I'm also not saying that we should stop looking for seniors. I'm not talking about going to the other extreme here. Just to bring right proportions, which as I think should be a bit more on the side of junior developers and skills other than coding. On the other hand, my advice to junior developers is: ignore gatekeeping the best you can. There are no real programmers in the sense that some other programmers are not real. If you work as a programmer, or if you participate in an open source project, or if you simply code something in your free time, you are a programmer. And you are real enough. You can also make mistakes and that doesn't take that title from you. If you see that your company promotes the hero culture or that your team is getting organized around one person, **schedule a 1:1 talk with your manager** and talk about it. I think it's important to make it a 1:1 meeting and just dropping a hint over a coffee is not enough and bringing the topic while the whole team is listening may result in denial and a pushback. Instead, Prepare for a longer talk, make sure that the manager treats it seriously, and bring up good arguments. If it doesn't work, if nothing changes, I guess that means you should look for a better job. And that's it.