Beyond the Commit, Episode 3: Sebastian Titze - Transcript

In this episode, we explain the essential challenge faced by every modern software organization: technical debt. We analyze how to define it, distinguish between conscious strategic debt and unintentional "mess", and, crucially, how to manage and eliminate it to maintain development speed and team morale.

Our host, **Paweł Dolega**, discusses this topic with **Sebastian Titze**, CTO at Corify, an experienced technology leader. Sebastian, drawing on his career experience from working in the financial sector to consulting and management, shares his strategies.

He explains how to measure technical debt using metrics like cognitive complexity and change frequency, emphasizes that fixing tech debt must be framed as a business decision (ROI), and stresses the importance of organizational culture and team motivation in addressing it.

Beyond the Commit is brought to you by VirtusLab & SoftwareMill. Our podcast spotlights CTOs and senior engineers, sharing candid stories that resonate with technology and business leaders alike. Expanding on our popular technology blog (45 k monthly readers), the series adopts a more personal, conversation-driven format.

Want to listen to the podcast on Spotify, Apple, or Amazon Podcasts? Check out the Beyond the Commit website for details. Below you'll find the transcript of the conversation.

Pawel: All right, so, uh, we are up. Uh, hi, Sebastian. It's great to have you here on the pod today.

Sebastian: Yeah, thanks for having me here. Very lovely.

Pawel: Yeah, so, today we are going to talk about the technical debt in the organization and all things around it: what it is, how do you solve it, how do you manage working with the code base, with the technical debt, and things like that.

So, I'm really excited to talk about this topic, which is very relevant to many different organizations and many different team leaders. Before we do that, I thought that it might be nice if you gave us a short introduction about yourself and about your story, especially as it intersects with the technical debt—how it happened that you are so interested in this topic.

Sebastian: Yeah.

Pawel: Yeah, thank you.

Sebastian: You will see, when I was preparing, I really saw that I had a lot of touch with tech debt. So, my journey started almost 30 years ago when I got my first computer. It really made me curious about doing some programming stuff there.

I taught it myself. It was in the time when the internet was not so common, so I had to buy some old notebooks and learn it. This helped me then during studies to get my first professional job in software development.

I worked for a small company—today we would say it was a startup—to create software for dentists. I had one pivotal point there: when I had to go to the customer, I talked to him, and he had just a different opinion on what he wanted to see there. But this sounded super interesting to me—what people do with technology. So, I stuck to this through my whole career.

Because of this, I moved to IT consulting in Frankfurt, slipping into the financial industry. My first two projects, when I think about it now, were all about handling tech debt, because they were big migrations of data integrations projects. This was great learning. I made my way there to project management, again dealing with people.

At the end of this time, I thought I'd like to go back to the north. This job was in Frankfurt; I come from Hamburg, Northern Germany. I went to the great city of Quickborn. It sits here in Northern Germany. It's pretty small, but it was the headquarters of comdirect.

Pawel: Mm-hmm.

Sebastian: ... which was the, or which still is the digital daughter of Commerzbank, which is the second-largest bank in Germany. And there, I learned tech debt from very many perspectives. They ran their own data center, so it was not just code; it was also some old hardware we replaced, cables, and so on.

On the other side, there were those very, very big compliance products which touched the whole company. It was a very interesting time. But I always had an eye on the new things; I liked to do new things. I then had the chance to switch to the innovation management department there. I got in touch with startups and managed a program where new companies were able to test their products against this big customer base of comdirect. That was also super interesting for me to see how they work, what their problems are, and different.

I then moved on to even more management stuff. I moved a little bit from tech to people because I worked in digitization for a wealth manager in Hamburg. This was interesting because it was not so much about the technology anymore; it was there to solve their problems. It was more about getting the people to work with this technology. So, I would say they didn't have tech debt, they had just no tech at all.

Pawel: Mm-hmm.

Sebastian: They had tech debt but not really how we see it now today. And now I work for Corefy here and have all the responsibility for all the people who work with technology and the technology itself. I have a very different view on this topic now since we are developing a product. We are a young company, so you still have to find a good balance between keeping a good track, maintaining the speed. But on the other side, of course, we want to grow. So, to wrap it up, I saw it from very different perspectives: from a developer, even when I was junior, and from the project management or consulting side where you have a different view on this. You just want your projects to be done, so tech debt is more like something you accept. And now, a totally different view again.

Pawel: Right. That's an interesting intersection of work experience in the corporate world and in the startups ecosystem. And, by the way, when I think about it, I think when we talk about tech debt, really banking and finance is one of the very fertile grounds where you can, you know, learn how to deal with it. They got digitized so early. They deal with important information, sensitive information, and financial data. So, they have a lot of those old systems even written in COBOL, things like that.

Sebastian: [laughs]

Pawel: And they are still working and making money, right? [laughs]

Sebastian: Yeah, yeah, you're right.

Pawel: Yeah. So, going into this topic of technical debt, I think that one of the things that we could do at the very beginning is to try to define the term and, you know, let our viewers or listeners know, how do we define tech debt? What do you have in mind when you think about tech debt? What is it for you?

Sebastian: Yeah. You will find a lot of definitions out there. It's good, right? So, for me, it's quite easy. It's the cost for all the choices you've made in order to make faster progress or reach your goals faster.

Sometimes it's difficult to differentiate between tech and product. So, for me, it's really just the tech side, right? It's just like skipping version updates or skipping some test coverage in order to reach your timeline. What I don't see there is all the functional debt or, like, that you, uh, intentionally remove some functionality in order to meet the deadlines.

So, this is tech debt for me. One other thing which you should differentiate—because you should handle it differently or see it differently as outsiders—is that tech debt is something you can do intentionally or unintentionally. Unintentionally is more about bad code. And this is not really managing tech debt. It's something you have to manage with the teams to bring them forward. But this intentional decision is what I define as tech debt, using it in order to be faster on the market.

Pawel: Right. So, that's an interesting thing. I think early, maybe 2000, there was an article by Robert Martin, a.k.a. Uncle Bob, and he discussed this thing which is pretty much in line with your definition. He sort of said that there is tech debt which is the result of the conscious decision that you are going to sacrifice quality or correctness for the sake of speed. And the other thing is you did something wrong or something which is incorrect because you just didn't have the experience. He called this unconscious debt or inadvertent tech debt.

The latter, he called just a mess. So, he defined technical debt as something that is consciously being done.

Now, when we put those... that's about the semantics. But from what you are saying, especially as you mentioned this sacrifice of correctness in order to get some speed, it's sort of in line with the business thinking. When you are running a business, you often need to make some sacrifices. And, in the world, especially in the startups when you have finite resources, being time, money, or whatever, you need to make some choices about what is more important.

So, it sounds like in case of the conscious decision, there is some value in the tech debt, right? You take some debt, but it is perhaps something that is, at the end, a reasonable choice. So, could you tell me a little bit about how you differentiate between this good and bad tech debt? Is there something like a good tech debt, or do you have in mind some sort of classification when you look at the code base and you see like, "This is something where we consciously do that and we know that we need to sort it later," and, "This is something where we just made a lot of mess and we have to deal with it?"

Sebastian: Yeah. That's a good question, right? Whether we take tech debt or not, and this directly leads for me to the question, how do we measure tech debt, right? And to the question of return of investment.

To answer this, I don't see that there is a real number you can put on tech debt that says, "We have five here," or so. What you can do with tech debt if you have to make this immediate decision—what we are talking about, this is not about maintaining—it's more about, "Can we do this, can we sacrifice this for this time to deliver faster?"

I think this depends highly on the estimation and the effects. It is about translating this debt to business impact, right? So, what does it mean for the future? If this helps us now, we gain this, but we lose this in terms of velocity or stability of the system. So, we can put at least a guess, a number for this, and how much it costs us to fix this later.

Once you have those two numbers—when you know what does it help us now, what it brings us from the business side, and how much does it cost us from the tech side to fix it—then you can do the calculation. This is done in the situation where you have to think about, "Can we take this tech debt for now?"

Pawel: Right. So, from what you're saying, putting this into the business mindset is very much, you know, first of all, it's a business decision, it's a trade-off. And the second thing is, contrary to the, I think, maybe not a popular belief, but the phrase "technical debt" is sort of negatively charged. But from what you're saying, it doesn't have to be in all cases. It's just a trade-off that you make to, you know, bring the business results.

Sebastian: Yeah. And also what we phrase as bad tech debt, this is not always intentional. Maybe the developers don't have the skills right now. But what is also often the case is that you don't know enough about the product or about the business, so that you build... that you may choose, in extreme cases, the wrong architecture. But still, in the moment you made the decision, it was a good decision. So, it's more a... it more shows that you have learned more about the system, right?

To be honest, I don't see only a very small amount of tech debt as real bad tech debt, like someone really did something wrong because of the lack of some knowledge or so.

Pawel: Right, right. I like your answer here about this, you know, the stage of the knowledge that you had at the time when you made decisions. I think we often forget about this when, you know, when engineers join the project and they look at something and they say, "Well, that was obviously done incorrectly. That was obviously wrong". When, in reality, more often than not, they were smart people making some decisions; they had limited knowledge at the given point in time, and they did the best they could at a given point in time with the knowledge that they could get.

So, yeah, that's an interesting point. Now, before we go, because obviously we wanted to talk about how do you deal with the tech debt , how do you remove it or how do you refactor the application or whether it's worth refactoring it. I think maybe first we'll talk about how you measure the tech debt in your code base or in your technical product. As in, as the first step in order to figure out what to fix, where to invest your time and money, you surely have to have some way of measuring or assessing particular areas of code in the context of the technical debt.

Sebastian: Yeah, yeah, sure. We have... first again, I don't think there is a single number, you know, like, "We have tech debt of 10," or so, and we have to go down to five. I think you have to take multiple measures or metrics or things into account.

What I do is I see there are three categories of ways you can measure it : the direct ones, like evaluating all the code; the second is more like measuring the whole development process and seeing if there is any connection to the tech debt ; and the last one is about talking to people about what their mood is, because this sometimes also has a good connection to it.

I can jump a little bit into those topics. The first one, what we do is kind of a hotspot analysis. When you look at tech debt, it's all about slowing down changes or slowing down future changes. So, the first thing you have to understand is what changes very often in your system. And it's not only maybe the code base; the code itself could also be infrastructure nowadays.

What we do is... and you can easily do this with Git—check how high is the change frequency of several parts of the system. And once you know this, you have to understand, "Is it difficult to change or not?"

Therefore, we use default tools like SonarQube. I think this is very well known. What we completely use is this metric of cognitive complexity that shows how complex is this code to understand and change. When you put it on an axis, you get a nice graph and you see everything on the top right that should really take some attention and should be reflected maybe in the future. So, this is the direct approach.

Pawel: So, just to... Before you jump to the second point, what you're saying is that one thing is the cognitive complexity—that's the one sort of the axis—and the other is the frequency of change. So, the idea here is that you want to look at things that are changed often and are at the same time difficult to, you know, cognitively comprehend. Because you don't want to invest your time in things that even if they are complex, they are rarely changed, and they don't bring much of the return of investment. Is that correct?

Sebastian: Yeah, exactly. Yeah. Because, you know, if the code is running, has no bugs, but is really ugly to read, then it's not really important to refactor it, right? Some people would love to refactor it, but, because everyone wants good code, right? But from the business point of view, it's not really necessary. So, yeah, you're totally right.

Pawel: Got it, got it. Makes sense.

Sebastian: Yeah. Okay. And so the next point is that you monitor your software development process and whatever approach you take there. We take a look at the DORA metrics. So, we try to deliver as fast as possible, as often as possible, and keep changes small. Whenever something drops, like the deployment frequency or the lead time to change, it could be a good indicator that there might be something wrong from the technical side, like you have to refactor the pipelines or so.

On the other hand, this is also risky because sometimes it's always the most easy choice to say, "Yeah, this is tech debt or this is something that the technical team has to do". Sometimes it's more complicated to fix it because it's kind of a cultural thing. So, you can use it, right, but you have to do this with care.

Pawel: Got it. So, in that case, you are looking at the things or, you know, in a dynamic perspective. You said that you are looking for the drop in certain metrics, like DORA metrics. So, you see that it might be a chance of tech debt being introduced in the recent changes.

Sebastian: Yeah, exactly.

Pawel: Mm-hmm. All right.

Sebastian: The last thing is not so direct, but I think it's very important. It's about, in all those conversations, checking how people talk about parts of the system. This is also important. Like, if people, for example, have this with the pipelines, they are super slow, and you hear it from different corners and from different people, then it might be also a very good indicator that you should refactor it.

For me, even if it does not directly affect the business or makes the process faster, it's very important that people want to work within the environment. So, it's important then sometimes to also say, "Yeah, let's improve this," right? So, the work is more smooth. This is the third and most indirect measure, I would say, how you can get out of this tech debt.

Pawel: Mm-hmm. So, it's more based on a feeling and sort of a morale of the team working on it.

Sebastian: Yeah, exactly, exactly. Yeah.

Pawel: Which is obviously then important, you know. Obviously, there is a lot... I think I found sometimes that the sort of how motivated the team is impacts the performance. Sometimes even a tiny investment in a certain area brings a lot of, for the lack of a better word, joy to the engineers and the entire team that finally something has been resolved. That might have only... this didn't affect the business entirely, but affected the sort of the mood and feelings of the development team.

Sebastian: Yeah, exactly. There is also some risk of over-engineering or bringing everything to perfection. But sometimes it's also important that people want to be heard, and as you said, to raise motivation. And it's more fun to work with that.

Pawel: Right, which brings us to another aspect, because as you said, there is a danger of over-engineering. In most of the teams, at least from my experience, there were always many different candidates for the areas of code to be refactored.

When you look from the business perspective, and you put a businessman hat on you and you think only from the business perspective, apart from the motivation, it was, at least for me, always typically very difficult to assess the return of the investment. Because my reality was, and the reality of people that I talked with about this problem is that most of the time it took significantly more time to actually fix the technical debt than it seemed at the very beginning. That included obviously the time to refactor the piece of code base. But also, the thing was that more often than not, it also brought other bugs and other defects to the software. Often, or at least sometimes, there was a long tail of those bugs that had to be fixed, that were introduced in the first place because of the good intention of, you know, improving the existing code base.

From that perspective, could you elaborate a little bit more about how you approach this ROI? You mentioned these two axes and where are the hotspots of things that you want to fix. But probably you want to have some sort of estimate whether that's worth it. Whether at the end of the day it's going to be profitable to actually spend some time on this piece of code to refactor it. Could you tell us a little bit about that?

Sebastian: Yeah, I mentioned it partly, right. So, first thing is, of course, you have to know whenever you change something in the system, some tech debt is connected with it, and it doesn't make sense now to tackle this topic. It only makes sense because the system is changing and tech debt is in your ticket system all the time. So, you can't just estimate all of them.

Therefore, you have to know, "This could be tackled now". Then you have to go into, as I said, you have to understand what's the impact of fixing this or of not fixing it in the current scenario. You make an estimation, you talk with the business about midterm goals of this functionality. Then you can derive a good estimation: does it make sense to fix this now? So, at least you have the business side of it.

I'm a big fan of doing this incrementally. So, not doing this big bang refactoring and refactoring everything. That's why this hotspot analysis is really valuable. You can see, "To what degree does it make sense to refactor this?" "Let's start with the most important things or most valuable things, and then see how this evolves". You can apply the same pattern you use in software development in general also with refactoring: just do small steps, and maybe this helps already a lot.

Pawel: Mm-hmm.

Sebastian: This can remove the risk of over-engineering because you have steps and steps and steps.

Pawel: Did you find yourself at some point in time in a situation that it was almost impossible or at least very difficult to make this incremental refactoring, and you had to do a complete rewrite with some other approach?

Sebastian: Not myself. But, as I said in the beginning, I was participating in two big projects where the whole thing was stuck. At least one of them was stuck because of some version updates. This was not so much because the code itself was, or the code base itself was in some kind of state where it could be further developed with acceptable costs.

It was more like the vendor... It was a migration from a closed source project to an open source project. The vendor had just updated the software to a new major version. The update of this for this project and all the customization would have been so expensive that it didn't make sense anymore. So, this was the time where the company was unable to maintain this anymore. For myself, not.

Pawel: Right. So, the-

Sebastian: Exactly

Pawel: ... the answer in that case was, you know, sort of restarting the project and doing it somewhere on the side from the very beginning with the new architecture or the new updates?

Sebastian: Yeah, exactly. So, the decision was to check how far the open source community was, that they had good solutions. They revalidated it. So, the cost for using a more common technology was strategically much better because you saved cost. From the people's side, there were a lot more developers who could work on this because this was not so highly specialized as the-

Pawel: Mm-hmm

Sebastian: ... original project. So, it was a really strategic decision there.

Pawel: Got it. Right. So, from the operational perspective, when we talk about... You already hinted at that about this incremental approach. But maybe we could dig deeper into the approach of solving the technical debt.

I'm really interested in two particular sides of it. So, first is, you know, how do you approach it from a technical perspective, or any tools? You already mentioned SonarQube, so that's one thing. Maybe there are others, some tools, some engineering approaches that you use. Things like, I don't know, canary releases with, you know, two versions running at the same time, some rollback strategies, things like that, depending on the nature of the problem, obviously.

That's one side of the whole question. The other side that we look at, you know, dig into later on is the organizational or the management perspective, or management approach to sorting out the technical debt. I'm thinking about things like some people choose to have one iteration, if we are following, for instance, some Scrum methodology. We do one iteration when we focus entirely on tech debt. The other organizations or managers have this approach: "Let's spend, I don't know, 20% or 10% of our time, or the whole of the time to constantly try to, you know, fix the tech debt".

So, let's start with this technical, I think, engineering and technical approach to fixing tech debt. Any pieces of information about that area?

Sebastian: Mm. You mean like tooling, right?

Pawel: Tooling, yeah, an approach as I said like things like, you know, canary releases-

Sebastian: Mm-hmm

Pawel: ... some rollback strategies, things like that.

Sebastian: Yeah. So, for us, it's highly connected, right? And it's a little bit hard to differentiate. Leaving out the program context, we use for tooling—and this is mainly for not adding up more tech debt—is what we use is different approaches. The first one is that we have code reviews.

Pawel: Mm-hmm

Sebastian: Even if, for example, a junior developer does some stuff, it's always the second developer that has to check what is done there. This is not about blaming; it's really about improving and getting a better understanding of the overall architecture of the system and also the requirements of the business in this case. So, this is one very important thing here.

The second thing, if things become bigger, we do the same with architecture decisions. We always write down every decision in an ADR (Architectural Decision Record). Once this thing is written down, everyone who is kind of touched with it, or who is just interested, has to at least check it. As long as there are strong objections against it, then it is not accepted. So, then the discussion goes on, right? So, we reduce the risk of making a wrong architecture decision. That's what we do on the organization side.

From the tooling side, I've never tried real tools. I know there are some outside, but we just use SonarQube for priority. Everything else... I could only recommend using the best IDEs, because they help a lot. And, of course, nowadays, you can't avoid the AI stuff, because it helps a lot doing easy refactorings, at least if you don't do complex stuff.

Pawel: Mm-hmm. So, this is more of an approach of preventing the creation of the technical debt in the first place, right? What you said about the reviews, making sure that you don't introduce it into the existing code base.

Sebastian: Yeah, exactly. Mm-hmm.

Pawel: What about the management or organizational approach? Is there anything specific, how do you approach this thing?

Sebastian: Yeah, this is more or less where all this is embedded. As I said, I really appreciate that approach to gradually deal with tech debt, because I think these sprints where you just have to do tech debt is really demotivating. Sometimes it also introduces new tech debt. And then you might end up with just different tech debt.

So, our approach is that we have, or try to establish, a culture of ownership and responsibility for the code. If you have to implement a new feature, it's not just that your job is to implement this feature with good quality. It's also to have a look on the left and then on the right. If you see anything you can improve, then do it.

We don't have a fixed amount for this, like 10 or 20%. It's more like everyone has a good idea about the code he is dealing with, and if it is to a certain amount, he has to fix this. The same is with bigger things. That's where this ADR approach comes into account, and people write this down. Also, doing implementation. In this way, I've found that it takes away this special feeling about tech debt. So, it's normal day-to-day work.

For those bigger things, because now we have them in the system, we have regular meetings where we discuss those things: how urgent they are now, and so on. Then we decide what to do with it. But again, we always try not to do a big refactoring, but also, how can we gradually introduce this to the business and to the product? So, we can slowly switch.

What we did now is, for example, we changed the whole architecture from a more, the front-end API architecture. Before that, we were proxying everything into this, and now we introduced a more gateway approach. We didn't do it for all of our services in one big bang, but we do it gradually, like for every service.

Pawel: Mm-hmm. I think that when we talk about technical debt, for most people what is interesting are those big technical debt areas, because those small ones are relatively easy to fix. It's not that difficult to sell them, another interesting topic, to the executives or the business-oriented people. Because technical debt, I think one of the difficulties with that, is like with many teams you can always argue, as you mentioned, that there are some things that need to be improved. The value from the business perspective is often difficult to assess, especially for non-technical people.

So, I think that part of those strategies comes from the need to explain to, for instance, executives or CEOs that we need to do something. Because when you think about it, it's like, "We've got to spend some time, we're going to invest some money, you're not going to get any new features from that". "It's like we'll do internal," and it sounds bad, because, "We screwed up in the past, and now we need to spend this amount of money, which by the way we are not entirely sure that it would be exactly that". More often than not, it will be more than we initially expected. "And we need to spend this amount of money to fix it".

So, I think that a lot of these strategies come from the fact that you need to somehow deal with and explain in a reasonable way that you need to fix the tech debt, and how to approach it. For instance, "The product development is not stuck for two months, we are only fixing tech debt, and not introducing anything that will improve our business in terms of new features and competitive advantage".

You, as far as I understand, respond to the CEO of the company. So, we need to also justify those investments in fixing the tech debt. Did you find it difficult in the past? Did you find some strategies working better than others? For instance, like I mentioned before, a fixed amount of time that is spent consistently on technical debt? Or you just, if you have a big chunk, especially a big chunk of work that is related to technical debt, you just divide it into pieces. And usually you didn't have any problems with, you know, with justifying this investment to the management?

Sebastian: Mm-hmm. Especially in the beginning, right, when I was arguing more about the benefits this technology has. What I've learned is that the key here is really about business impact. So, what value does it have?

Even if it is a bigger chunk, changing from one technology to another, you have to translate it into this language the business understands, and this is always a business impact. In the end, from my point of view, you cannot just change the technology just because of the technology, right? You must always name the reasons behind this. Like, and if it is only, "We can't find any people who can maintain this anymore," this might also be a strategic reason, right? But this is then not because the new technology is cooler.

This is often the case that sometimes people tend to talk about the benefits of a certain technology, but you can't do this right, especially if you talk about bigger chunks, because this is always about an investment. And always people want to get something back from it.

Pawel: Mm-hmm. So, it seems like a tall order in general to justify it in a reasonable way. That's my experience, because when you fix something, when you try to fix something, you see the burden that it has today, and that's clearly visible today. You see how it is in terms of people's motivation, finding people that want to work with it, and even speed. What is unclear is the amount of time that is going to take to fix a given thing. The other thing that is unclear is how much will it improve the given situation in terms of the motivation or, for instance, speed.

So, this, I think, comes down to how do you reliably estimate the impact of the given thing in terms of the coming back to this ROI and the, as you said, the explanation. If you say that, "We are going to fix this thing, and it's going to improve the speed of introducing new features, for instance, which are affecting the business". Did you find it difficult in the past? Do you have some ways to assess it in terms of what would be the outcome, what would be the benefit? It seems to me like it's a difficult thing in general.

Sebastian: Yeah, it definitely is. Again, what you have to do, you have to find some business partners who really help you in forming the near-term future ideas—what you can do with it—not in a technological way. What does it mean for the product?

So, first, you can estimate how much it will take me to replace this chunk of code or to fix it. You can also hardly, but somehow, have an idea what does it mean in terms of the processes you have to change, or the monitoring, whatever you build around this. What is important is what it means for the product. So, what perspective do you have if we do this from the product side? You have to find some companion, you know, in order to argue about it or to win these discussions.

Again, I mean, it has changed something. Tech departments are no longer only a cost center, right? They are also there to enable the business to bring up new features. This is more where I think you should act. Not so much in, "We need some money. I'm sorry, we messed up". It's more like, "We do this now. So, this means this for our product, for our business". So, this is more the direction I go there.

Pawel: Got it. Well, I guess, you know, it depends on the business. I think in your case, when the goal of the business of Corefy company, as far as I understand, is building the digitized market for insurance, the nature of it is pretty much technical.

Sebastian: Yeah, exactly.

Pawel: There are many other businesses for which the technical part or the technology is, as you mentioned, the cost center that allows to do often the physical business, for instance. So, I get it. In which case it might be a little bit more difficult because it goes back to the awareness of the senior management and executives of the role of technology in business.

Sebastian: Yeah, exactly. I think so too. But this has changed, luckily, at least.

Pawel: Yeah, definitely. I mean, it's difficult these days to imagine a new business that is being formed, that is... all the businesses rely on technology more than in the past. And I think this awareness is becoming more and more widespread.

From this, did you have any experience, for instance, what I saw in some of the... in my experience, in others people's experience is that there is a chunk of work that is a technical debt, and we know that it's bringing some problems. Maybe it hampers the velocity of development, maybe it brings down the motivation. And we decided to tackle it, probably not picking too many battles at the same time. So, we wanted to pick one or two things and then spread it over some periods of time, maybe a quarter or two quarters, and divide it into manageable chunks.

Sometimes what happens is that, you know, we sort of thought it's going to take a quarter, a few tasks here and there. In the meantime, doing some new features that were requested. The quarter has passed, and we realize that we are nowhere near the end of fixing what we planned to fix. So, did you have any experience with such situations? How did you deal with them?

Sebastian: For me, this is also an iterative approach, right? If you say you want to do this and that, different chunks of code in a certain amount of time, then you still get some value by just doing the most important one, which you started with at the beginning. So, I was never in the situation where... I mean, when it's okay to not continue with the things planned. So, yeah, not really.

I told you about those two projects, but those were more like writing those big code bases, and there was a lot more stuff and it took much longer. But this was a whole different approach. So, I've never experienced something like this where you had small chunks and you didn't finish it. In the end, to be honest, for me now, it wouldn't be that bad because at least you have finished the most important ones.

Pawel: Mm-hmm. So, that is at least, you know, one sort of piece of advice that comes from that: if you divide it into certain chunks, be sure that you pick the most important ones at the very beginning. Because even at some point, you know, it happens that for some reason, either the stakeholders or the shareholders of the project lost patience in a way to invest in the thing, at least you fixed the most important thing. You don't have to finish everything. If you fix like 60% of the pain, it's still 60 percent of the pain alleviated, which is still a success, right?

Sebastian: Yeah, or maybe it's even more, right? If you prioritize right. So, um, yeah.

Pawel: Right. So, do you have certain areas... or do you have an experience with certain areas of the code base that you know that this is a messy code base, there is a lot of technical debt, no one wants to work with that, but you know that it's very low on the priority list? And some people... engineers are not motivated to work on that, understandably. Do you have experience with such areas? That's just the fact of life that in most projects that are mature enough, there's going to be some place which is not fun to work with, and that's just the fact of life. You have to deal with it.

Sebastian: Yeah, yeah, you're right. The question is, is this in one of those areas which has to be refactored, you know? Now let's imagine it is. Then what I found out is that it is most important to give everyone the chance to really change it, right, and not just to have to work with it. So, this is maybe the difference.

Sometimes you want to change it and then you have to say the business doesn't want you to change it now, right? They have to go on and go on. From my point of view, as a tech leader, you have to prevent this. You somehow make sure that everyone gets the chance to fulfill this responsibility I talked about and really have the chance to change it. Then it is not so bad anymore because, yeah, you have to change the code, but you know better, and you have the chance to change it. This brings some motivation to the people. It's really worse if you can't change it but you want to and have to work with it for some reason.

Pawel: Mm-hmm. Got it. So, maybe, coming back to this original... one of the first questions that we had at the very beginning about the good and bad tech debt, and the tech debt being the result of the conscious decision of the trade-off. Let's say that you are making this trade-off, and you are choosing a certain piece of the software that you are building to be suboptimal. Do you have any techniques or advice in terms of how to contain it? You know that you are going to do suboptimal work. How do you prevent it from spreading over all other places or being unmanageable in the future? Do you have any experience with that?

Sebastian: What do you mean exactly?

Pawel: Like, what I'm thinking is that in many organizations, or that was at least my experience, you work with something, and there is a pressure to deliver something. The pressure makes it so that it's very difficult or impossible to do good engineering work. So, you make some sacrifices. The problem often is that, as they say sometimes, you make those sacrifices and they are, in a way, eternal. You never go back to it, and you just created new technical debt that is unmanageable in the future.

Sebastian: Oh, now I understand. For me, that transparency is key. You have to make sure that everyone knows, "We have a certain amount of tech debt in the system," meaning we have to deliver as fast as we can. But we also have to make sure that we can pay back the tech debt or the debt.

We do this like, whenever we see there's a new feature to be made, and we make this decision as you said, and make some compromises, we write it down into a ticket. You can't really estimate it now because the system is changing. But what you can do is you can at least—this is mandatory for us—put a rough T-shirt size on it. Say, "Okay, this is a really big chunk, and we have to do this".

You also have to write down how it affects the system in terms of impact for the business. For example, what is nearly a no-brainer is if you say, "We have to fix it because otherwise we will have continuously some security things," for example. With these two numbers—the impact and the size—you have to make them transparent.

Of course, at the end, if someone decides, "Yeah, I'm not still interested," but that's all you can do, I think, to make really transparent what the code base looks like and how stable it is.

Pawel: Got it. So, it's again, a conscious decision made upfront. Is there something like a broken window theory of technical debt? The broken window theory is like, I think that's the economic theory that, you know, there was a broken window. When people found something that was unmaintained as a broken window, they were more eager to, you know, start littering around that and not care about the general state of it.

What I'm saying is that you have a certain piece of code which was maybe even deliberately suboptimal, and it was written in a bad way. Then it spreads because people start to build on top of it another and another thing. Given the shaky state of this piece of code, it gets expanded and expanded, and it's even more difficult to handle it in the future.

Sebastian: Yeah. I mean, if you miss this—what I've said, all this, the reviews and so on ... then yeah, I think this might be a good candidate to rewrite this whole functionality. Because, you know, if the base is so unstable that everything which is on top of it, you have to change anyway, then. So, um, uh-huh.

Pawel: Mm-hmm. So, it's again, sort of the responsibility of everyone in the system. You rely on people reviewing the code base and trying to put some checks in place to avoid the expansion of the technical debt from one place to the other.

Sebastian: Yeah, exactly. You should notice, right? Sometimes you don't. But at least it's not... if it is only, as I said, if it's not changed very often, it cannot become such a risk, right? But if it is, then you have again and again these reviews or these mechanisms to check it. If you monitor the stability or the error rates, whenever something is changed in this piece of code, and it takes much longer every time, then, of course, you have to be alarmed and check this again.

Pawel: Mm-hmm. Got it.

Sebastian: Mm-hmm.

Pawel: So, switching maybe to something else, to a topic that is more right now the main topic in the internet and pretty much the entire world, which is AI. Maybe before we go into "Is there any way how we could use the AI or generative AI in particular for fixing the technical debt," maybe the first thing is, do you find any differences in terms of the effectiveness of using the AI tools for development in the legacy code bases, or the code bases which are ridden with technical debt, compared to the other ones, which are maybe more stable, more properly engineered?

Sebastian: To be honest, I have no experience with it. Because I'm new, I've not faced anything like that.

Pawel: Mm-hmm. Do you find in general, you know, I'm talking about things like Cursor or CoPilot, do you find it effective in your production code bases?

Sebastian: Yeah, definitely for the easy things, right? It helps, you know, if you have recurring things you have to build again and again, then it helps, really helps, to be much more productive. You don't have to write the 24th controller or persistent structure. In easy cases, that helps a lot.

Second is that it also helps, and maybe then we have this connection to the legacy code base, it helps to explain even complicated code passages, not the whole architecture and especially not the function. But it helps to understand single functions, for example. That helps a lot as soon as we use it.

Pawel: Right. And what tools are you using as of today?

Sebastian: Yeah, so, CoPilot we use. The guys are relatively free, right, as long as it is not shipped to some, some China or so. We use Cursor, we use Cloud Code, and I don't remember the third tool. Windsurf, I think it's called.

Pawel: Oh yeah, Windsurf, yeah.

Sebastian: Yeah, we use those tools.

Pawel: Right. So, maybe, I think we touched a lot of different areas. Maybe before we wrap it up, maybe you could tell us, you know, if you were to give two, three sentences of advice for an engineering manager or a CTO that maybe joined another company and has to deal with the code base littered with technical debt, is there anything that you would advise such a person?

Sebastian: I should advise some old man, right? I have three topics.

First one is really what I talked about: trying to make tech debt part of the day-to-day business work. The job for you as the tech leader is to establish this culture of really making everyone able to fulfill this responsibility and to take care of the code. So, it becomes more part of the daily work. This is basic for me.

The second thing is, and this might sound a little bit different, you should sometimes be bold and just delete old tickets in the system. If they don't appear often, then they can't be that important. If they are really important, then they will come up anyway. So, you get some air to breathe. Really focus on the important things.

The last one... I thought about it, but it is more important than ever: "Stay hungry, stay foolish," because AI is changing everything so fast. If you don't have an eye on this development, I think you will lose track and fall behind very fast.

Pawel: Mm-hmm. That is an interesting thing. All of what you said was very interesting. But the last thing, it also sparked one of the ideas in my mind. Maybe in the not so distant future, dealing with the technical debt would be entirely something that could be postponed. Because at some point, maybe the AI would be able to rewrite certain pieces of software in a more, you know, better-defined way or more according to the sort of technical specifications, less architecture.

I'm talking about this specifically because I've talked with several technical leaders recently who had a big refactoring technical debt in their project. So big, in fact, that when they tried to assess, it was like, "It's going to be two or three years of work". "We are unsure whether that's worth it". At the same time, "We see people are very unmotivated to work on this". "It takes a hell of a lot of time to introduce new features". But then if you put in two or three years of work, it's a lot of time. When I think about it, I don't know about your opinion, but when I think about the estimate of two or three years, in my mind it might as well be five years. Who knows? It's very difficult to assess two or three years of work, right?

Sebastian: Yeah. Exactly. I mean, it depends on how well you know this code base, right? But normally, you know, you start working on it, and I've never seen it differently that you learn more about the system, and normally it takes longer. So, yeah, I think this makes the decision even harder. Right? I mean ... maybe they already estimated one day, one year and said three years because they know, right? But I don't know about that. [laughs]

Pawel: Right. Right.

Sebastian: Yeah.

Pawel: Sure. So, do you think it's going to be... Is it something that you foresee in the future such as refactorings or rewrites ... Because this is often moving something from one old technology to the other. You want to have it more or less functionally similar. But for some reason, you use very outdated technology, and maybe you want to move... Let's imagine you have a custom orchestration software in your organization. You really think these days, well, if we moved it to Kubernetes, at least it would be standardized.

There are many different problems. You have custom software which is difficult to maintain because no one knows it, only in your company it's being used, so no external people know it. It's difficult to recruit people. There are a lot of different bugs that need to be fixed by yourself because that's your proprietary technology, and all those other things.

In that case, it seems like it's maybe not automatic, but it's something that you want to keep similar features, just re-platform it to something else. Do you think in the foreseeable future—I'm talking about two, three years from now—would it be something that might be doable by AI tools?

Sebastian: Yeah. I have two different views on that. The first one is that it will definitely enhance and become better, and definitely change the way... I mean, we see this now already. The second thing is more like, I'm wondering if this will be three years. What will the code base of the future be, right? Because, for me, it's like, today, in a very simple way, you have the business, they have a problem to solve, and they go to a developer and they talk about it. Then they find a solution, and the developer then translates it to some high-level language, and then the machine can somehow execute it, right? So, you always have steps.

Now, when the AI can really do this translation of the code base to another technology or code base, then I wonder if somehow the code base will be really something only a developer can understand, you know what I mean?

Pawel: Mm-hmm.

Sebastian: I really wonder. I think if this really is possible, then I think we will have more like deterministic prompting or extended prompting where business and someone more technical work together. Then I don't think that we will have these languages anymore, which only the developer can understand and read.

Pawel: Mm-hmm. It sounds like, from what you're saying, it sounds like a more distant future. It doesn't sound like something that happens in two, three years. I think the progress in technology is really amazing in general, but then at the same time, many things really take years, right?

Sebastian: Yeah, yeah. We're on the top of the hype curve right now, so the reality comes down. But still, I think it will help, right?

What I wanted to say is, if AI can really put everything from one code base to another, then we are at the stage where it can somehow understand the whole business. Then we are on this stage more. So, yeah, I can actually see that.

Pawel: Right. Right. And realistically speaking, I framed this question in a way of moving or re-platforming from one technology to the other, keeping the feature set the same. But realistically, it's never the case. Whenever you re-platform something because of some technical constraints and because of how business changes, you usually do it a little bit differently, even from the feature perspective, even from the business perspective, right? That is my experience.

Sebastian: Yeah. I mean, you make technology choices based on some business requirements, right? If you choose a new technology, then some things will be done differently. You don't want... if you translate Java code to Go, you don't want Java code in Go, right? You want Go code in the same style.

You will not just... you can run a monolithic architecture in a Kubernetes cluster that doesn't really bring all the advantages this has to bring. This is really hard to do by an AI today, right, because this is so connected to the business and all this context.

Pawel: Right. Right. It's like the business impacts the technology that you pick, but also the technology that you have sort of impacts how the business is working to some extent.

Sebastian: Yeah, yeah, yeah.

Pawel: All right. So, I think we may wrap it up. It was a very great experience to have you here and to pick your thoughts on this topic of technical debt. Thank you for being here. Hopefully, we'll have a chance to meet together again at some point and discuss other interesting topics.

Sebastian: Yeah, I would really appreciate it. Thank you very much for having me here again. Thank you.

Pawel: Thank you.