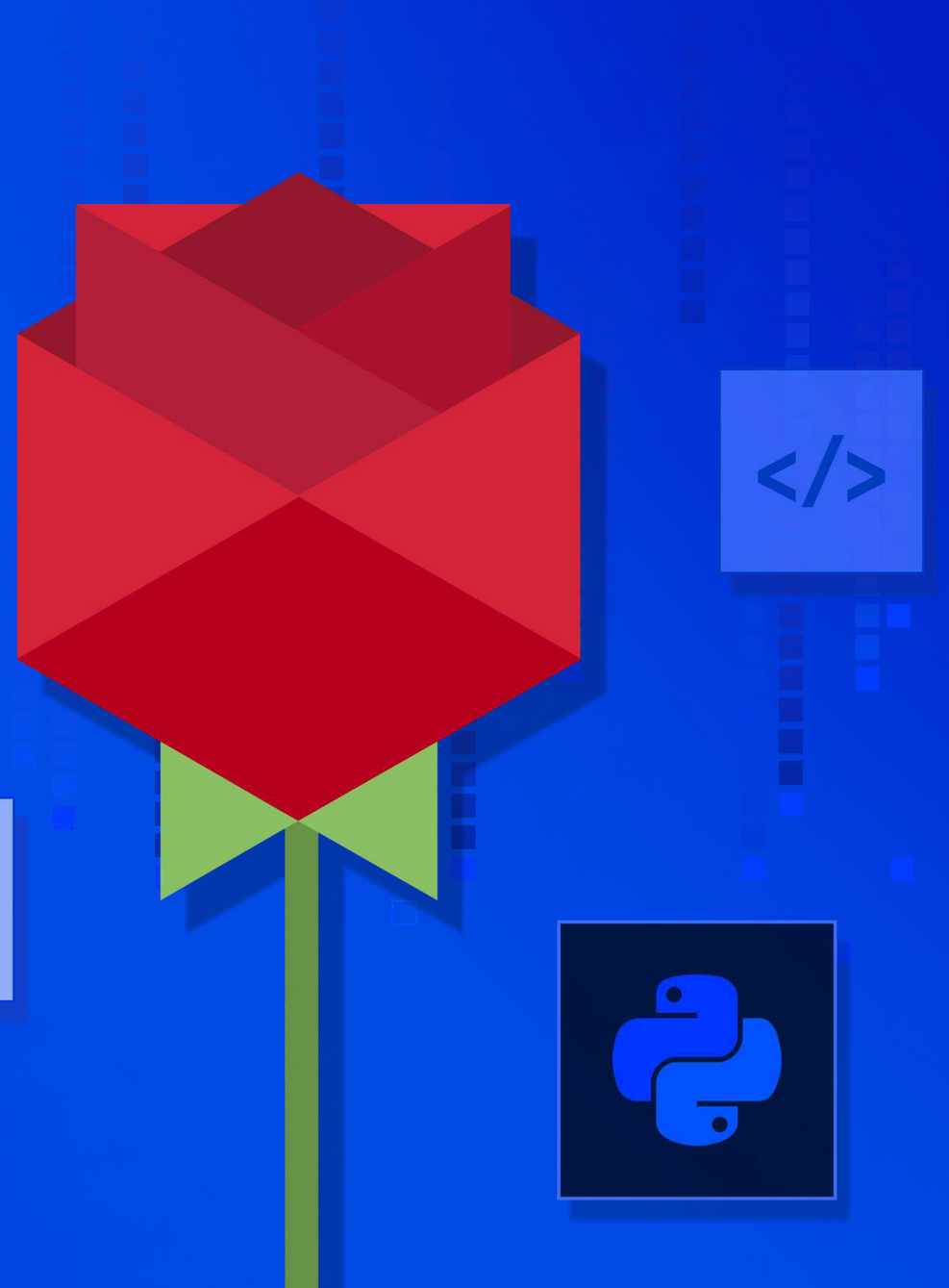
VINTA

Tech Debt Tango

Dancing Through Bad Code and Refactors with Grace



WRITTEN BY

Hugo Bessa

Principal Engineer

Intro

Managing technical debt poses a significant challenge for software developers. The most common struggle is determining the best time to address known issues. It can be even harder to fit solutions into tight product backlogs full of shiny new features.

We developers often find ourselves bothered by messy code, mainly when legacy code lands in our laps. The urge to start afresh runs intensely through our veins when we encounter a piece of code written ***who-knows-how-many*** years ago, lacking quality criteria and documentation.

Years of experience mitigate the impulse to rush headlong into things: dealing with legacy code is part of the job, and it's rarely possible to avoid it. Before investing significant time and resources into tackling these challenges, an experienced professional better assesses the value generated by delving into these snake pits of life.

But the big question is: How do we convince Product leadership that the time to delve into the snake pit has come? This is especially true when Product leadership is formed by non-technical individuals. And more, how do we measure the exact value of a technical debt solution before implementing it?

In this ebook, I've compiled insights gleaned from my decade-long tenure as a software engineer on **prioritizing technical debt and effectively communicating its significance to stakeholders**. Upon reading this resource, you'll acquire valuable knowledge on seamlessly integrating debt management into your team's everyday practices, preventing it from becoming an unaddressed issue.

Enjoy your reading!

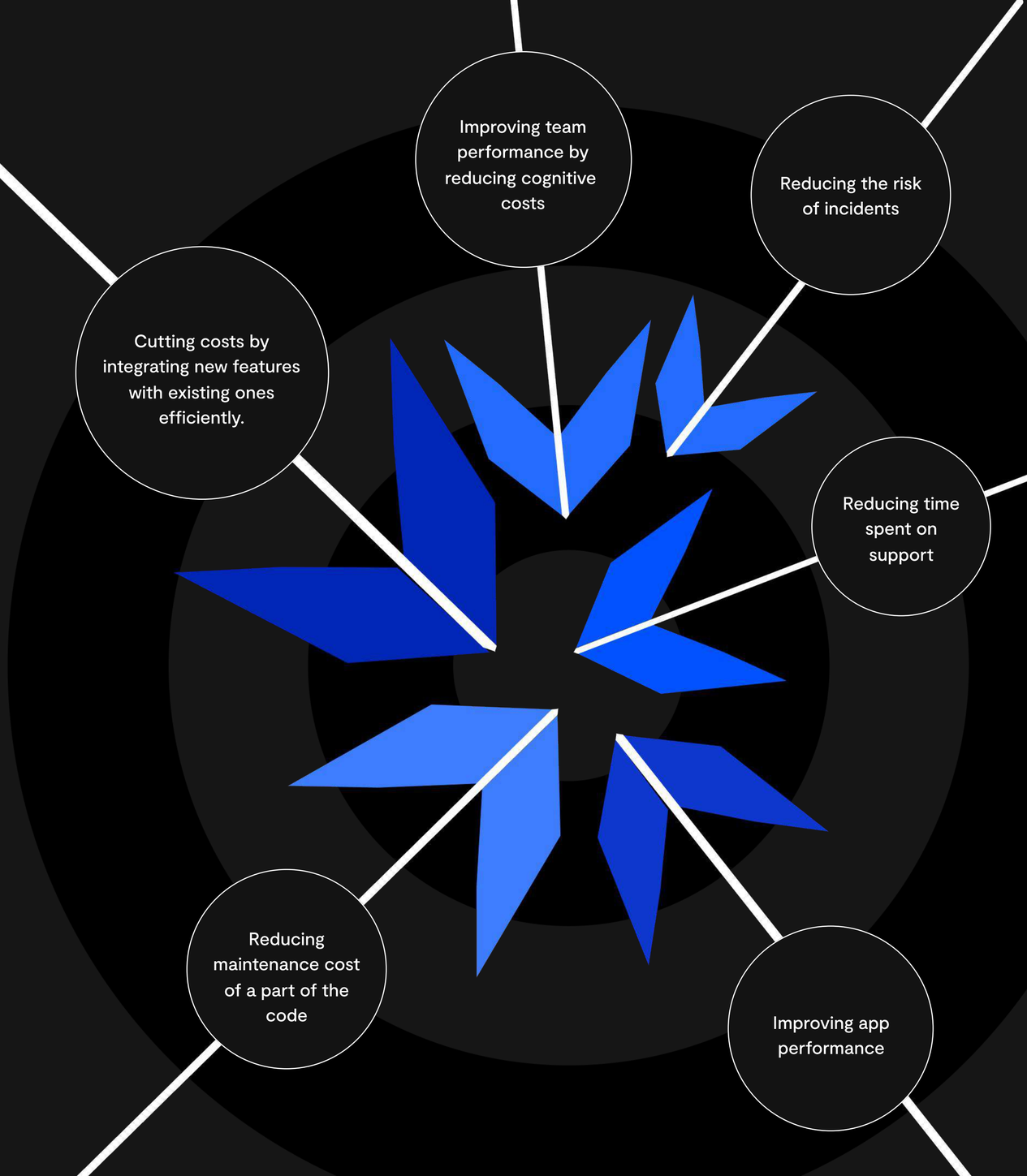
Hugo Bessa
Principal Engineer at Vinta Software

Strategically Integrating Technical Debt Management:

Priorities, Metrics, and Outcomes

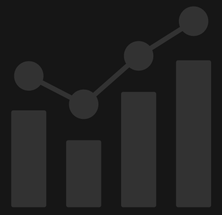
A technical debt solution must have **clear objectives** that must be included in your strategy. With defined goals, measuring progress, allocating resources effectively, and ensuring alignment with overall project objectives becomes easier.

There are several objectives you can choose from, such as:



Engaging with leadership is crucial to determining the priority objectives during your team's strategy definition. By understanding these priorities, you can accurately assess the current state of each objective and establish clear goals to achieve them. This makes prioritization easier and ensures that efforts are directly aligned with the strategy.

Measuring the results will give substance to the practice and reveal the impact of "invisible work" on the strategy. This not only fuels the morale of the technical team but also helps to gain the leadership's trust regarding technical matters. There are many popular ways of measuring team throughput and stability, **for example**:



DORA metrics for DevOps



Team performance: Delivery Time, Review Time and Deploy Time



Tracking code reviews quality: Number of comments in PRs, number of changes per PR;



Tracking outdated dependencies: LibYears.

All those metrics can be used as arguments for prioritizing technical debt solutions, but they don't come for free. It takes considerable work to measure them and compare their statuses before and after changes. In addition, it's often impossible to forecast what results to expect before implementing changes.

We might have a pleasant surprise, significantly impacting key indicators; however, there's always the possibility that the outcome will be different from what we initially planned, resulting in wasted time on something that didn't yield significant results. Nevertheless, there are ways to reduce this risk, such as involving **more action** and **less strategy**.



Mitigating Technical Debt:

Practical Insights and Implementation Strategies

Filipe Ximenes, Vinta Software's partner, once made an analogy that I took for life: he compared software developers to chefs.

When you order something in a restaurant, a lot happens behind the scenes before the meal reaches your table. Chefs might master many practices and processes to guarantee we have the best experience possible, practices and processes we had no idea existed.

For instance, a chef knows that to avoid cross-contamination, **they cannot use the same pans and utensils for different foods**. As a customer allergic to seafood, I trust all precautions are being taken in the kitchen so that my Chicken Parmesan does not come with residues of shrimp from the Seafood Stew. I don't directly ask the chef to wash the pan or to use a different one; I just trust they will do that.

This also applies to software development: Developers, particularly those more experienced, have various practices and processes that are abstract to people from other areas. This gives them the responsibility to ensure that the software is built with the requirements for the product's success, such as developing tests, investigating bugs, implementing improvements in the development environment, documentation, and refactoring, among others.

All these activities impact development time and should only be done with criteria. As developers, it is up to us to know when to include them in our day-to-day work, seize opportunities, and ignore them without many side effects.

When we discuss larger technical debts, we face a more significant challenge. Including in our routine is even harder without impacting the project timeline. In these cases, we have to break debts down into smaller parts and use tools such as **Proof of Concept (PoC)** and other experiments to prioritize, define efforts, and estimate gains.

These experimentation and evaluation activities usually require little effort and can be done daily without asking anyone for permission. This is part of our work as much as it is for the chef to wash their hands before cooking. And the chef doesn't ask for permission to wash their hands.`

These tools give us inputs to argue with leadership and include greater efforts for technical debt solutions in the product strategy with confidence in their effectiveness.

In summary, **it's essential to perceive the evaluation of technical debts** as a discipline within our profession. We should integrate it into our routine to build confidence in executing larger projects that may not directly relate to the product's functional aspects but still align with the overall strategy.



Technical Debt Isn't Necessarily Bad



Measuring the impact of software quality and productivity can be challenging. Yet, studies like "**Code Red: The Business Impact of Code Quality**" by Adam Tornhill and Markus Borg provide compelling evidence. According to their findings, investing in software quality yields significant benefits because it leads to fewer defects, faster delivery times, and increased customer satisfaction.

So, how do we justify introducing technical debt into product development? I believe the keyword here is "opportunity."

There are many unproven hypotheses and opportunities in sight (which are usually the main motivations for developing the product) — especially in the early stages of a product.

Investing excessively in quality during these early stages can lead to higher long-term development costs. However, it may also result in delays in releasing the product, causing it to miss the optimal time to market. In such scenarios, it becomes imperative to carefully evaluate the trade-offs and determine which sacrifices can be made to capitalize on immediate opportunities, recognizing that certain implementations may need to be revisited.

Known concepts like Clean Code's **YAGNI** ***(You Ain't Gonna Need It)*** and **KISS** ***(Keep it Simple, Stupid)*** reinforce that premature optimizations and abstractions can undermine time-sensitive opportunities and create unnecessary complexities in your software.

Opting for simplicity and implementing only essential features typically prevents an increase in the team's cognitive load and maintains productivity. Yet, you should be careful about how deep you should go regarding the application's future prediction and your attachment to specific architectures and solutions.

Frequently, the advantages of having a product ready sooner can yield crucial value for reinvestment despite awareness of potential future challenges with the introduced code. This code can subsequently be refined with a larger and more capable team. Failing to be pragmatic about what is essential for the product's success can result in its discontinuity.

Prioritizing Technical Debt in Practice

As mentioned earlier, prioritizing the resolution of technical debt requires that it either be affordable enough to integrate into our daily workflow or closely aligned with the product strategy. To identify who is who in your backlog, it is necessary to pay attention to the following factors:



Impact on team productivity and cognitive cost

Complex architecture and legacy code can hinder tasks. Prioritize technical debt based on its impact frequency and complexity.



Security Impact

Consider vulnerability severity and likelihood of exploitation when prioritizing fixes for product debt vulnerable to attacks or malicious intent.



Impact on maintenance

Debts that generate data inconsistencies or problems in important user operations, especially when these occurrences require manual work to be solved, are strong candidates for prioritization. The frequency of occurrences and the time taken to find the solution are critical factors for the debt solution to be taken seriously



Scalability impact

if there is an intention, pretension, or forecast of a considerable increase in the number of users or the use of specific functionality, the impact of debts that affect performance should be evaluated. Tools like load and stress testing are essential to understanding the impact and prioritization of technical debt solutions.



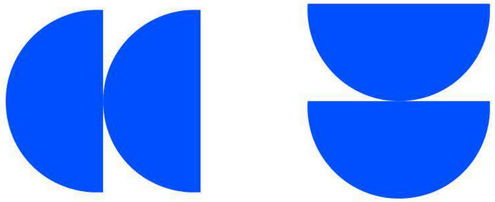
Development Cost

Technical debt solutions with low costs can be implemented between activities or during pauses during lengthy tasks. However, those with higher costs should be broken down as much as possible into smaller deliveries, each with its impact assessments. Activities that can no longer be divided should be estimated so that the cost is considered in prioritization.



Opportunity Cost

When working on a problematic part of the code, considerable effort is involved in understanding these problems and thinking about possible solutions. When this time is spent for whatever reason (whether it's an activity that integrates with the problematic code or even the solution to a bug), there is an opportunity to solve it without recurring this cost again. We are only sometimes available to do so, and the opportunity cost only sometimes justifies immediate prioritization. Another opportunity is when we know we will develop something that uses code with technical debt, and we can reduce the development cost if we solve the technical debt beforehand.



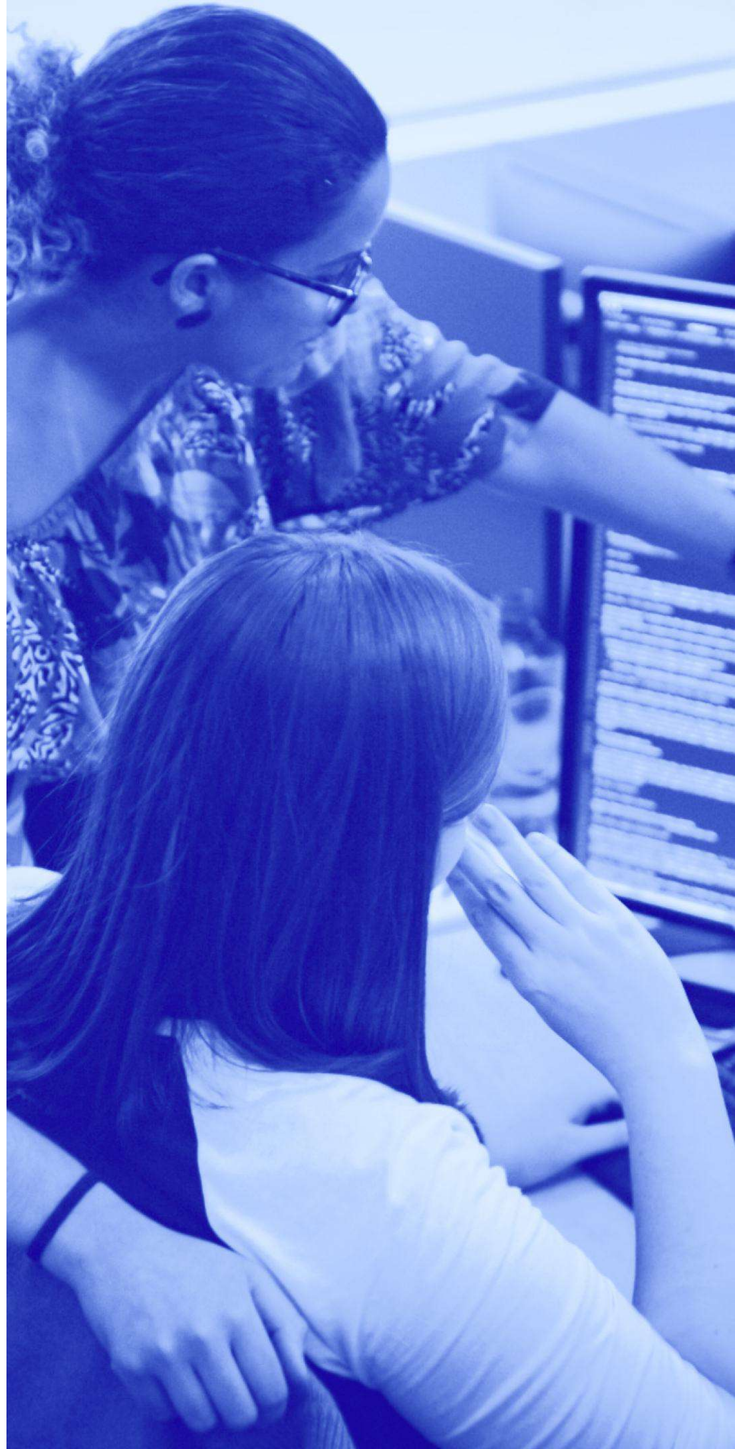
The combined impact of these factors will determine the importance and urgency of addressing technical debt. However, the degree of consideration given to each factor will depend on the product strategy.

A healthy way to prioritize activities is to **create a formula that considers the importance of each factor** for your product and a numerical measure that represents the level of impact or cost of each.

The numerical measure can be more factual, like the number of estimated days to complete the activity, or it can be more abstract, for example, 1 representing low cost, 2 representing medium cost, and 3 representing high cost. Whatever works best in the context of the product and the team.

Special attention is recommended to keeping dependencies updated. This can influence your application's security and significantly impact your team's performance with almost no effort. This performance improvement can occur because, with updated versions, we have access to new functionalities that can simplify the use of dependencies and enhance the development experience. The effort is low because implementing these features would be much more costly than updating the libraries.

Performing this analysis routine will make technical debt prioritization more manageable and integrated into your team's daily workflow. Additionally, articulating concerns, risks, and opportunities aligned with the strategy becomes more straightforward, employing pragmatism and presenting arguments easily digestible even for non-technical stakeholders.



Prioritizing Technical Debt: Communicating with Product Stakeholders

The most challenging aspect of handling technical devs is persuading non-technical stakeholders of the need to revise outdated code. There are various techniques for allocating time for the development team.

Throughout my career, I've observed teams utilizing the Portfolio approach, which allocates a specific percentage of the workforce to focus on making improvements. This method can be effective for products with numerous small to medium-complexity issues, as these can be addressed weekly or monthly. However, I've identified **two significant challenges** that could make this dedicated time for addressing technical debt problematic:



The **flexible definition** of Technical Debt

It's typical for non-technical stakeholders to fit features and support tasks into the technical debt bucket. This makes us use the technical debt portfolio for other things with a more visible impact on the product strategy, while the actual technical debt doesn't get its place in the backlog.



The **product development pace** cannot decrease

When your team has more significant issues to look at, like architectural changes or module replacements, the Portfolio approach tells you to accumulate time without working on technical debt and then use the accumulated time to fix it. This strategy looks OK in theory, but reducing the speed for product updates may disengage part of the user base and create pressure for the product managers to deliver faster. In practice, you might not even have the opportunity to utilize the accumulated time due to shifts in priorities or the relentless pace of ongoing tasks.

In general, you'll end up dealing with the same issue of not having saved time: convincing the leadership of the importance of revisiting legacy code. The difference is the negotiation would now involve a lot of mental gymnastics and stretching of the product rope.

The best approach is to not negotiate at all, but instead just be transparent about the benefits of investing in technical debt and share the risks of not doing so. This means more work because your team needs to assess the impact of each task (if it's big enough, of course), but it also means you'll speak the same language as the leadership: **product strategy**.



Conclusion

When dealing with technical debt, it is essential to find a balance between introducing and resolving these "problems." Although it is natural for developers to be bothered by legacy code and to want to rebuild it from scratch, it is vital to consider the impact on the project's time and resources.

The solution to technical debt must be aligned with the team's objectives and priorities. It must take into account aspects such as productivity, maintenance cost, security, and scalability, but also consider development and opportunity costs.

Ultimately, evaluation, prioritization, and eventually, the solution of technical debt should be considered disciplines of the software development profession. By including them in your daily workflow, you can have confidence in executing larger projects that may not be directly related to the product's functionalities but are fundamental to its long-term success.



How you communicate the importance of investing in Technical Debt solutions is crucial for prioritizing it. You must ensure you speak the same language as the other stakeholders and are aligned with the product and team's strategy. **Being transparent and sharing risks is vital to making other people (especially non-technical) understand the value of your proposals.**

With a balanced approach based on objective criteria, technical debt can be managed, and the product's quality and continuous success can be ensured.

About the Author

Hugo Bessa is a Principal Engineer at Vinta Software, where his expertise in web development has led to significant contributions across numerous projects. With a solid foundation in Computer Science, Hugo has dedicated over a decade to mastering the art and science of software development, specializing in the Python programming language and its popular framework, Django. His work spans the entire development stack, from intricate backend systems to dynamic front-end interfaces, showcasing a deep commitment to crafting scalable, efficient digital solutions.

Beyond his technical skills, Hugo is recognized for his thought **leadership** within the tech community. He has shared his insights and experiences through engaging talks at prestigious events like **Python Brasil** and **PyGotham**.

Hugo's dedication to continuous learning and his passion for sharing knowledge make him an asset to Vinta Software and a valued mentor and speaker in the broader web development community.





About Vinta Software

Vinta Software is a team of software consultants, and developers prepared to turn digital ideas into market-ready products. Among the companies that benefit from our software are some well-established tech giants, like Airbnb, Salesforce, and Shopify.

Vinta supports businesses in realizing and scaling their software aspirations. For that, we like to use Django, React, Python, and JavaScript — a stack for which we are specialists and a market reference. Yet, we have an open mind to experiment and understand each project's needs.

We are also up for the task if you need to fill skill gaps in your company or scale your team. With our Staff Augmentation service, you can speed up your product delivery while maintaining top quality and meeting your roadmap expectations.

Connect with us and get to know more about how to accelerate digital product development!

Visit our website: www.vintasoftware.com

Our Tech Stack

django NEXT.JS python node.js React aws

Acknowledgments

Thank you to Flávio Juvenal and Djeefther Souza for their invaluable time and expertise in reviewing this article. Their thoughtful insights, constructive feedback, and attention to detail have contributed significantly to its quality and clarity.

Email
contact@vinta.com

VINTA