yeti.co

THE TECH DEBT CHECKLIST

If you notice any of the following issues within your development process, it's likely that you are dealing with tech debt.

FREQUENT BUG FIXES AND PATCHES

- ● Are you spending more time fixing bugs than developing new features?
- ● Are the same issues reoccurring despite previous fixes?

SLOW DEVELOPMENT AND DEPLOYMENT

- ● Do features take longer to develop due to complex code dependencies?
- ● Are build times excessively long, causing delays in deployment?

OUTDATED TECHNOLOGIES

- ● Does your project rely on outdated frameworks, libraries, or tools?
- ● Does upgrading dependencies feel risky due to potential breakages?

INCONSISTENT CODE QUALITY

- ● Are you working without established coding standards?
- ● Does your codebase consist of varying styles and conventions?

HIGH CODE COMPLEXITY

- Does your codebase have intertwined components that make it difficult to understand or change?
- Does it lack of modularity, requiring extensive modification for minor changes?

POOR DOCUMENTATION

- Is your documentation is outdated, incomplete, or non-existent?
- Do you struggle to understand the code without consulting original developers?

DIFFICULTY ONBOARDING NEW DEVELOPERS

- Does the codebase's complexity/poor documentation hinder productivity for new hires?
- Does onboarding require extensive knowledge transfer?

RECURRING TECHNICAL ISSUES

- Do the same technical issues reoccur, without a long-term solution?
- Do you frequently use workarounds instead of proper fixes?

HIGH MAINTENANCE COSTS

- Is a large portion of your budget allocated to maintaining the existing system?
- Is the cost of adding features/making changes disproportionately high?

BUSINESS AND TECHNICAL MISALIGNMENT

- Are there conflicts between business goals and the system's technical limitations?
- Has your ability to meet market demands or implement new business strategies slowed considerably?

LACK OF AUTOMATED TESTING

- Has a lack of automated testing practices led you to use manual, error-prone testing processes?
- Do deployments often introduce new bugs?

LOW CODE REUSABILITY

- Has a lack of reusable components/modules caused you to duplicate code in multiple places?
- Do you often rewrite code rather than leverage existing functionality?

PERFORMANCE ISSUES

- Does your app experience frequent performance issues, such as slow load times or high latency?
- Are performance problems often traced back to tech debt in the codebase?

USER COMPLAINTS AND POOR USER EXPERIENCE

- Do users frequently report bugs, crashes, or performance issues?
- Has the user experience been negatively impacted by technical limitations or unresolved issues?

YETI PRESENTS

NAVIGATING TECHNICAL DEBT: A GUIDEBOOK

TABLE OF CONTENTS

Navigating Technical Debt

In the fast-paced world of technology and software, time is of the essence.

We often see teams in a frenzied rush to launch their products, operating under the belief that, in order to be successful, their app needs to be the first to seize current trends. But does being first always mean being best?

In reality, the quest to be "first" often conflicts with the essential need for long-term success. Compromising best practices and taking shortcuts to launch will inevitably result in technical debt - a serious threat to your product's ecosystem... and your success.

The following are just a few of the impacts that tech debt can have on your project:

- *Reduced productivity*
- *Increased developer onboarding & ramp up time*
- *High risk of bugs*
- *Increased developer turnover*
- *Negative user feedback + reviews which hit the company's bottom line*

in this guide, we'll delve into technical debt, exploring its causes, far-reaching consequences and, most importantly, effective strategies for mitigating and handling it. Join us as we explore how seemingly minor decisions made in the heat of project deadlines can, over time, evolve into major challenges.

What is Technical Debt?

In the IoT software development process, technical debt is an issue that emerges when developers opt for quick solutions or take shortcuts to meet pressing business demands - typically at the expense of optimal code quality or system design.

These shortcuts, akin to financial debt, accumulate interest over time, gradually complicating the software landscape. With each workaround, complexity mounts, resulting in a codebase that is increasingly difficult to decipher, maintain and scale.

While these workarounds may provide a temporary reprieve in meeting deadlines or delivering features promptly, its long-term repercussions can be significant. Elevated costs, reduced productivity, heightened risk of errors and user dissatisfaction all become potential pitfalls that can threaten the longevity and competitiveness of software projects.

The following is a simple example illustrating how tech debt can occur, and the challenges it can create:

A development team faces a tight client deadline for completing an app project. Rather than request a deadline extension, the team works towards completing the project as quickly as possible, which ultimately requires them to take a few coding and testing shortcuts.

*While these shortcuts enable the team meet their deadline, they also lead to a convoluted codebase and several bugs that are quickly discovered through an influx of negative user feedback. Unfortunately, fixing these issues requires an additional developer to decipher and repair the confusing codebase, ultimately delaying the release of an important new product feature.**

The Impact of Tech Debt

While design and development shortcuts might seem like a quick fix for meeting tight deadlines, it’s crucial to understand the negative impact they can have on your development journey- and your end user’s experience.

In the previous example, coding shortcuts used to meet a tight deadline resulted in bugs and a negative user experience - but there are often more far reaching consequences. A codebase laden with tech debt can make in nearly impossible to implement new features without significant effort, in turn leaving the development team with far less less capacity for exploring new ideas, building new features and adapting to evolving market trends and technological advancements. With endless bugs to fix, the development cycle grows increasingly longer, and the time between shipping new features does as well.

It’s important to understand that, as time progresses and the codebase grows, technical debt tends to worsen and become more apparent. Because tech debt accumulates interest over time, each new feature added or modification made without addressing underlying issues further compounds the debt - and the effort required to repay this debt grows exponentially, often surpassing the initial time saved by taking shortcuts.

Technical debt also tends to amplify existing challenges and shortcomings in the software. Performance bottlenecks, reliability issues, and usability challenges become more pronounced as the codebase grows and accumulates debt, affecting the user experience and hampering your product's competitiveness in the market.

For all of the reasons listed about, tech debt can have a devastating impact on user experience. Apps laden with these issues often suffer from performance issues, reliability concerns, and usability challenges, leading to lower user satisfaction and reduced trust in your product. And, in todays extremely competitive tech landscape, an inability to consistently deliver new features can quickly lead users to other products that meet the expectations.

Types of Tech Debt

Deliberate vs. Inadvertent Technical Debt:

Technical debt is often discussed in two distinct contexts: deliberate and inadvertent. **Deliberate technical debt** involves a conscious choice by developers or teams to prioritize immediate goals, such as meeting deadlines or swiftly delivering features.

Inadvertent Technical Debt arises unintentionally, often as a result of factors like limited experience, or an incomplete understanding of the long-term implications of design and implementation choices. Developers may inadvertently incur technical debt by overlooking best practices, neglecting code quality, or failing to anticipate future scalability or maintainability concerns.

Design Debt:

Design debt encompasses the compromises and shortcuts made during the creation of a product's architecture and systems. As these systems mature, they often become more difficult to extend, maintain, or scale, displaying increased complexity and rigidity that hinder adaptability.

When scaling a product and introducing new features or requirements, the inherent flaws in the system's design become more evident, slowing development velocity and impeding the team's ability to respond effectively to change. What initially may have seemed like a cost-saving measure or a deadline-driven shortcut can lead to higher operational costs, decreased reliability, and a compromised user experience over time.

Code Debt:

Code debt represents the trade-offs made at the code level during software development, often stemming from the temptation to opt for quick fixes or shortcuts rather than implementing proper, sustainable solutions.
These compromises can lead to the accumulation of technical debt within the codebase, resulting in code that is challenging to comprehend, modify, or debug. This complexity arises from practices like copy-pasting code snippets, neglecting code refactoring, or disregarding established coding standards. Each instance of code debt adds to the overall burden of maintaining the software, as developers grapple with tangled, convoluted code that becomes increasingly resistant to change over time.

Test Debt

Test debt emerges when software development teams overlook or delay the implementation of proper testing practices throughout the development lifecycle. The consequences of test debt ripple across the software, manifesting as undetected bugs, diminished software quality, and heightened regression issues.

Without robust testing in place, bugs and defects may slip through the cracks, leading to user-facing issues and eroding trust in the software's reliability. Moreover, the absence of thorough testing makes it increasingly challenging to maintain and enhance the software over time.

As the codebase evolves and new features are introduced, the lack of adequate testing exacerbates the risk of introducing regressions or unintended side effects, further complicating maintenance efforts.

Documentation Debt:

Documentation debt arises when essential documentation, including requirements, design specifications, or user manuals, is absent, outdated, or incomplete. This can hinder comprehension of the system's functionality, design rationale, or usage instructions, resulting in confusion among developers, users, and other clients. Without clear documentation, understanding how the system operates, it’s intended behavior, and how different components interact becomes challenging, leading to inefficiencies and potential errors.

Moreover, inadequate documentation poses obstacles during the onboarding process for new team members, who may struggle to grasp the system's architecture, business logic, or implementation details without comprehensive documentation to guide them. This not only prolongs the ramp-up time for new team members but also increases the likelihood of errors and misunderstandings as they navigate the codebase.

Strategies for Managing Tech Debt

For all the reasons listed above, managing technical debt is a critical aspect of the software development process. At Yeti, we have a comprehensive approach to managing technical debt, throughout the software development lifecycle

.

Proactive Strategies

Agile Development Practices:

At Yeti, we employ Agile methodologies to oversee all our software projects. The Agile process is an iterative development cycle, which means that we operate within one to two-week sprint cycles, completing, testing, and receiving feedback on each piece of work before proceeding to the next. This structured approach inherently minimizes the accumulation of technical debt, ensuring that our codebase remains clean, maintainable, and aligned with project goals.

Continuous Integration and Deployment:

Continuous integration (**CI**) and **continuous deployment** (**CD**) practices automate the process of combining code changes and putting them into live environments, allowing for quicker feedback loops and delivering smaller changes that are easier to handle, CI/CD pipelines help keep the code in good shape and prevent debt from piling up.

Code Reviews

Code reviews involve team members examining each other's code before it's integrated into the project. This process encourages sharing knowledge, enhancing code quality, and spotting potential technical debt at an early stage. By tapping into the collective expertise of the team, code reviews act as a barrier against less-than-optimal coding practices and help maintain clean, maintainable codebases that adhere to established coding standards (you can take a look at our best practices repo for more information on code reviews and pull requests)

This structured approach inherently minimizes the accumulation of technical debt, ensuring that our codebase remains clean, maintainable, and aligned with project goals.

Continuous Integration and Deployment:

Continuous integration (**CI**) and **continuous deployment** (**CD**) practices automate the process of combining code changes and putting them into live environments, allowing for quicker feedback loops and delivering smaller changes that are easier to handle, CI/CD pipelines help keep the code in good shape and prevent debt from piling up.

Code Reviews

Code reviews involve team members examining each other's code before it's integrated into the project. This process encourages sharing knowledge, enhancing code quality, and spotting potential technical debt at an early stage. By tapping into the collective expertise of the team, code reviews act as a barrier against less-than-optimal coding practices and help maintain clean, maintainable codebases that adhere to established coding standards (you can take a look at our best practices repo for more information on code reviews and pull requests)

Automated Testing:

Automated testing involves having a program check your code to ensure it’s working correctly before deployment. Unit tests, integration tests, and end-to-end tests detect mistakes and ensure that any new changes won’t break anything that was previously working. These tests give quick feedback on any changes made to the code, confirm that the software behaves as expected, and make sure it's reliable and stable. This helps prevent bugs that can cause technical debt, saving time and effort in the long run. Take a look at our automated testing guide here.

Automated Testing:

Automated testing involves having a program check your code to ensure it’s working correctly before deployment. Unit tests, integration tests, and end-to-end tests detect mistakes and ensure that any new changes won’t break anything that was previously working. These tests give quick feedback on any changes made to the code, confirm that the software behaves as expected, and make sure it's reliable and stable. This helps prevent bugs that can cause technical debt, saving time and effort in the long run. Take a look at our automated testing guide here.

Reactive Strategies

Scheduled Refactoring

Scheduled refactoring sessions are regular opportunities for the development team to tidy up their code and fix any issues that have built up over time. This allows them to proactively address any accumulated technical debt by systematically improving code, enhancing the systems design, and eliminating any bugs.

Technical Debt Backlog

A technical debt backlog serves as a prioritized to-do list for developers, focusing on cataloging and ranking debt-related issues. By transparently documenting and prioritizing these issues, the development team can systematically integrate the most critical ones into their regular task list, ensuring that technical debt receives the necessary attention and resources for resolution.

Regular Assessments:

It is crucial for development teams to conduct regular tech debt assessments, such as code quality analyses, architectural reviews, or performance audits. This allows the team to identify and quantify existing debt within the codebase and make informed decisions about prioritizing those fixes.

Communication and Collaboration:

Effective communication and collaboration among team members and clients are essential for managing tech debt successfully. At Yeti, we foster a culture of transparency, openness, and shared responsibility that allows us to collectively identify, prioritize, and address technical debt, and ensure that it remains a visible and actionable aspect of our development process.

Tech Debt Best Practices

Navigating technical debt is an inevitable aspect of the IoT software development journey, and adopting best practices is a crucial for success. The following are a the steps we've taken to foster a culture of responsibility, continuous improvement, and success within our development team.

Establish a Technical Debt Policy

We've instituted a clear technical debt policy that lays the groundwork for managing debt effectively, and which outlines the guidelines and procedures our team uses to identify, assess, and address technical debt within projects. By establishing criteria for categorizing debt, defining roles and responsibilities, and implementing processes for debt management, we're able to proactively tackle debt-related challenges and minimize its impact on project delivery.

```
This example of
  Single::ToString( ),
  Single::ToString( String* ),
  Single::ToString( IFormatProvider* ), and
  Single::ToString( String*, IFormatProvider* )
generates the following output when run in the [en-US] cul
A Single number is formatted with various combinations of fo
strings and IFormatProvider.

IFormatProvider is not used; the default culture is [en-US]:
  No format string:                 11876.54
  'N5' format string:               11,876.54000
  'E' format string:                1.187654E+004
  'E5' format string:               1.18765E+004

A CultureInfo object for [nl-NL] is used for the IFormatProvider:
  No format string:                 11876,54
  'N5' format string:               11.876,54000
  'E' format string:                1,187654E+004

A NumberFormatInfo object with digit group size = 2 and
digit separator = '_' is used for the IFormatProvider:
  'N' format string:                1_18_76,54
  'E' format string:                1,187654E+004
Press any key to continue . . . _
```

Encourage a Culture of Ownership

Cultivating a culture of ownership instills a sense of accountability and responsibility among team members. By empowering individuals to take ownership of the codebase, including its quality and maintenance, we've fostered a collective commitment to mitigating technical debt. We encourage open communication, collaboration, and peer accountability to further reinforces this culture, driving continuous improvement and ensuring that technical debt is addressed proactively rather than being left to accumulate.

Provide Training and Support

We've ensured that our team members have the necessary skills and resources to identify and address technical debt by providing regular training sessions, workshops, and access to relevant resources that allows our developers to understand the implications of technical debt.

Celebrate Successes

Recognizing and celebrating successes reinforces positive behaviors and fosters a culture of continuous improvement. Whether it's acknowledging successful debt reduction efforts or highlighting team achievements in delivering high-quality software, we believe that celebrating success helps motivates and inspire our team to remain committed to building quality products.

• • • • • • • • • • • • • •

At Yeti, we recognize the importance of managing technical debt effectively, and we're committed to partnering with you every step of the way. If you're beginning your software or IoT software development journey, and would like to speak with an experienced team, feel free to send us a message, we'd love to chat!