STRV

CONTINUOUS OPTIMIZATION

Resolve & prevent issues before they threaten your business

WWW.STRV.COM

INDEX

Introduction

You've gotten a project off the ground. You've navigated the variety of tools available and know what you're doing. Maybe you're already armed with a successful product and are looking to keep that competitive technological edge. But perhaps you've noticed a decrease in velocity when developing new features, and performance is dropping. Sooner or later, refactoring becomes imminent. And that tends to be a bittersweet process. So, how do you approach it in a sensible way that keeps your team happy?

On the one side, we have engineers who feel unproductive as they explore archaic codebases and dependency integrations. Some even advocate a complete rewrite as the only possible action to take before the whole thing crumbles. Sound familiar?

On the other side, we have stakeholders who feel that rewriting is a wasted effort. After all, what is there to stop the new codebase from following in the steps of the old one and becoming obsolete in just a few years, or even months? Not to mention all that money already spent on building the current solution. With these common concerns, convincing higher-ups to allocate money towards a new solution can prove difficult.

As with everything, the answer is somewhere in-between. That's where this guide comes in.

Our ebook acts as a tool to help you identify common issues, their symptoms and their business impact, along with tips on how to improve your current codebase. We achieve this by using the Continuous Optimization strategy, which is ideal for feature-oriented products in active development—and where a complete rewrite would introduce an excess of duplicity in keeping both versions up-to-date.

STRATEGY

There are 4 core aspects of evaluating project health: **Technology, Architecture, Setup** and **Tech Debt.**

Introduction

Every project includes mistakes. The cause could be hacky code during tight deadlines, a lack of experience with the given technology or a malfunctioning crystal ball when proposing an architecture. Without a proper tech debt strategy, these untreated problems snowball into larger issues.

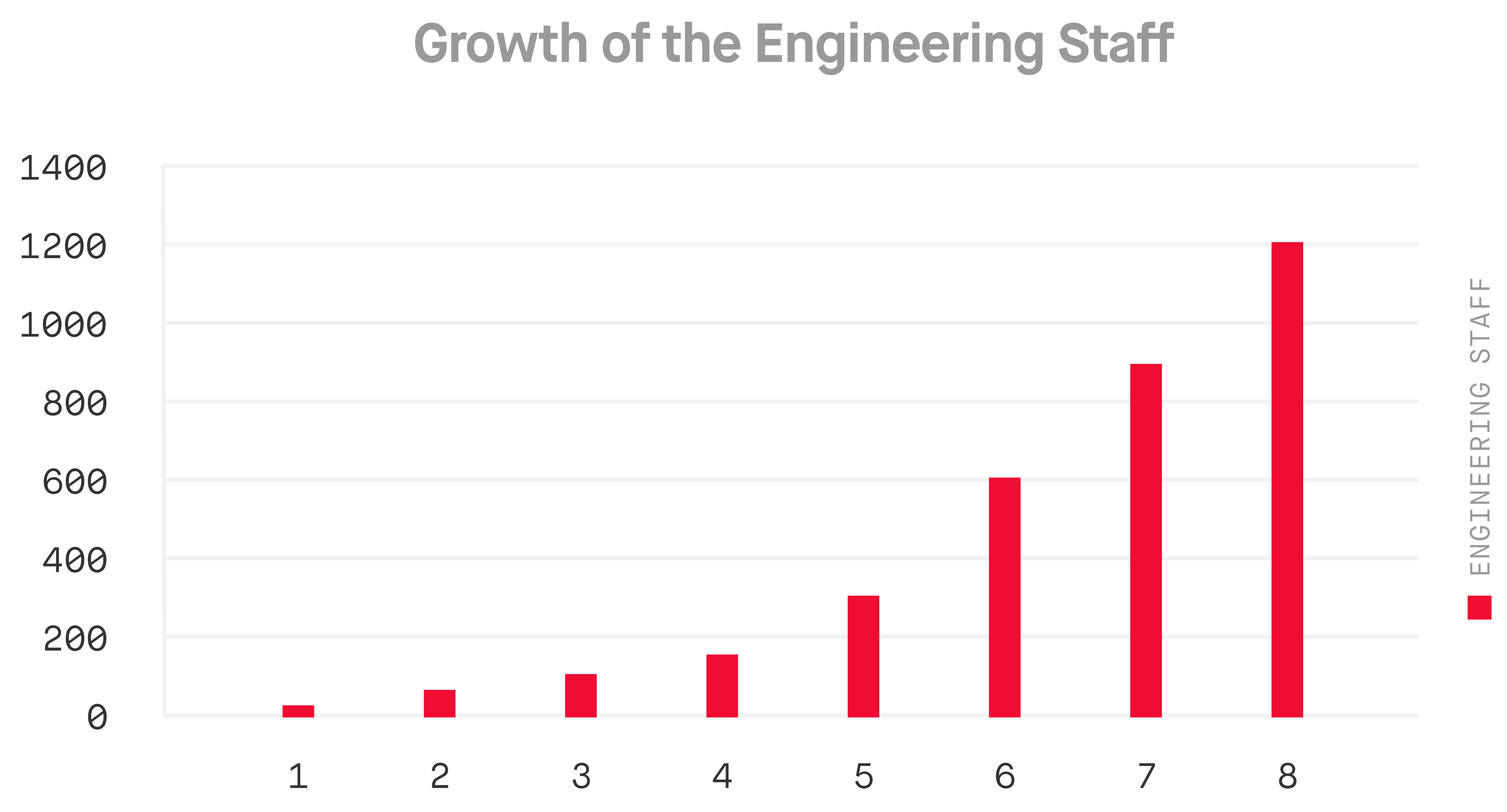
Any given project usually has one or more issues. They may not pose a problem right now but, if left unchecked, it may prove troublesome to resolve those issues once they start heavily affecting the business.

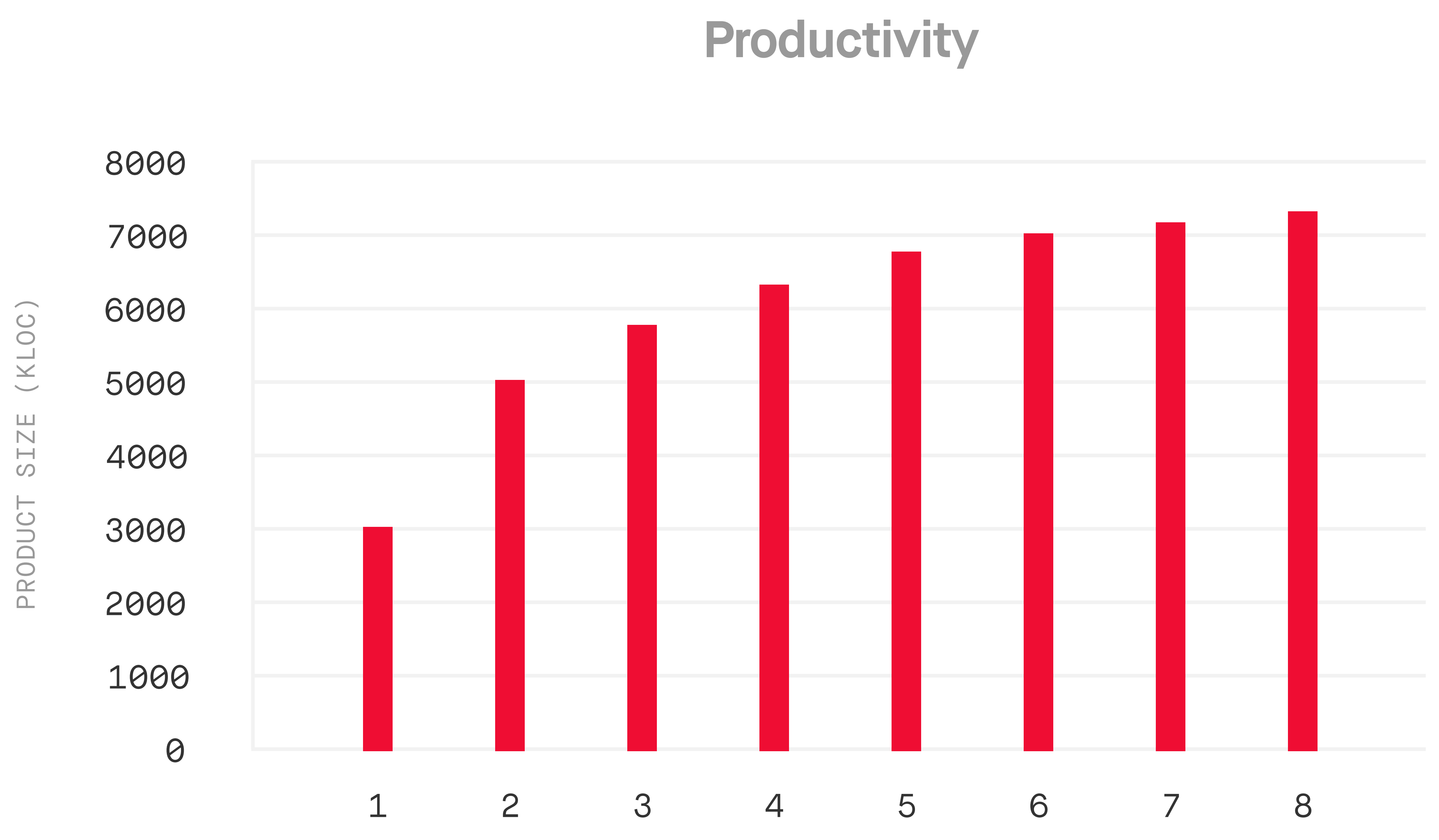
With proper utilization of Continuous Optimization and carefully set up processes, we’re able to resolve problems before they become a significant threat to your business. To explain what kind of threats you should aim to avoid, let’s take a look at a case study that outlines specific issues and how they can escalate.

Case Study

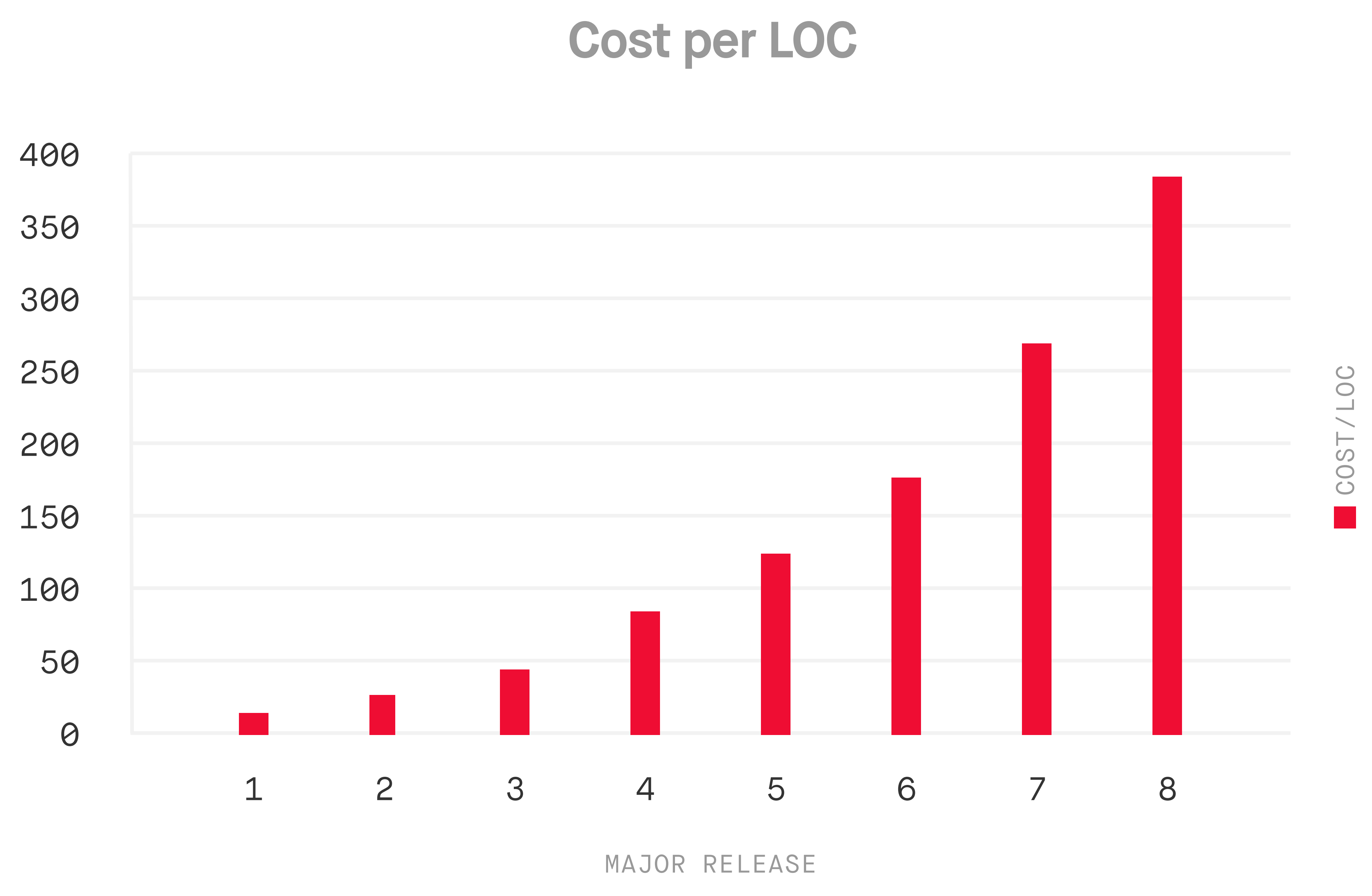
This case study (from the book Clean Architecture, by Robert C. Martin) utilizes data from an anonymous company. It looks at the level of effort needed to meet customers' needs. If the effort remains low throughout the system's lifetime, you're on the right track.

The first thing to notice is the growth of the engineering staff, compared to the company's productivity.





You'll notice that something's wrong. Although every release coincides with an increasing number of engineers, the code growth staggers. The next graph shows something even more worrisome.

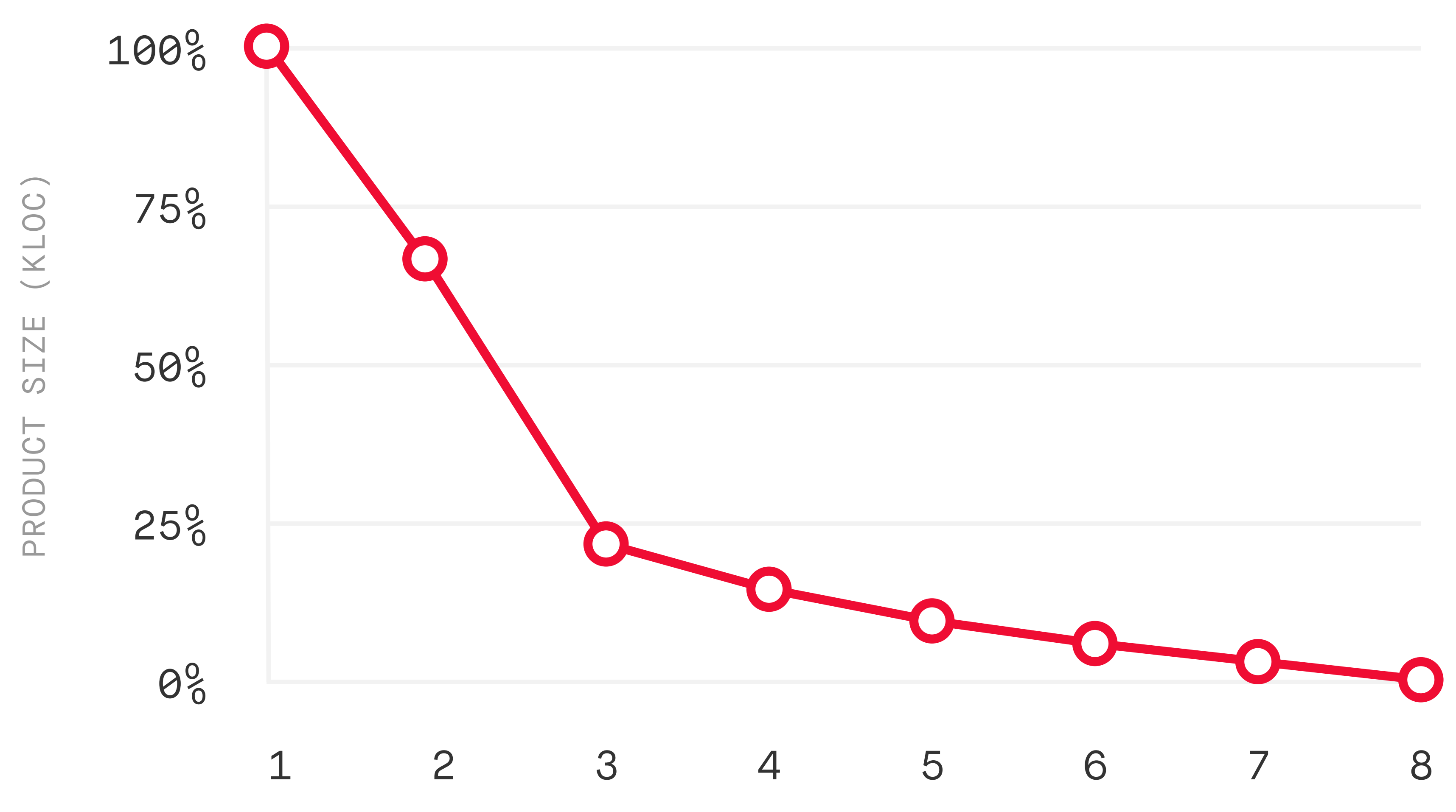


We can see that the cost per line of code has changed over time. All of this isn't sustainable. No matter how profitable the company, curves like this will drain profit, driving the company into the ground.

What caused this change in productivity? Why was the code 40 times more expensive to produce in release 8 as opposed to release 1? It's simple. Systems were thrown together without thinking ahead. The number of engineers was the only driver of output. Clean code and proper design structure weren't prioritized.

The following graph shows what the curve looks like to engineers.

Productivity/Releases

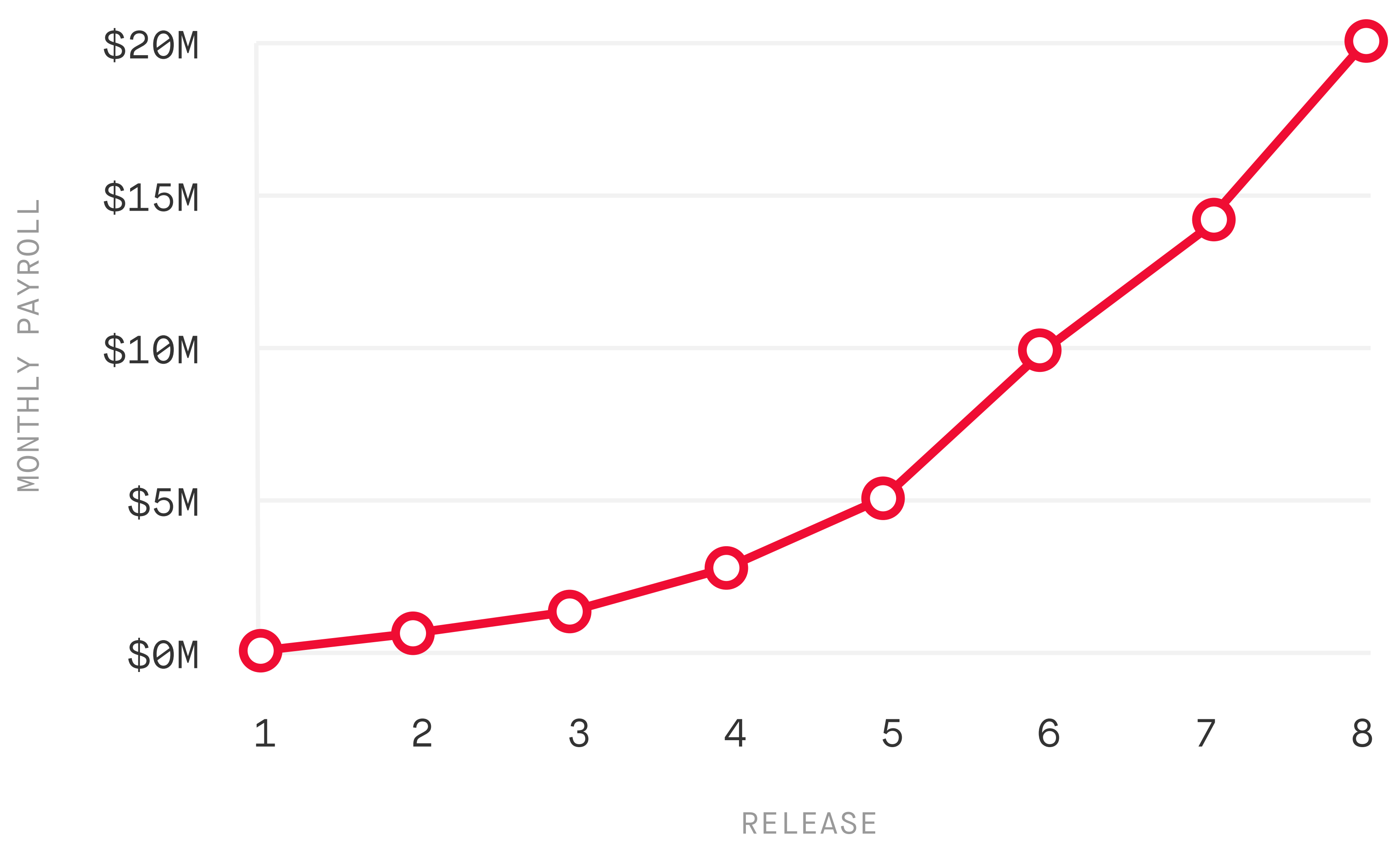


While the engineers started out at almost 100% productivity, that declined with each release. By the fourth release, bottoming out in an asymptotic approach to zero became the obvious trajectory.

The thing is, the engineers were still working hard. The effort hasn't decreased; instead, it's been redirected, moving away from features and focused on managing the mess.

Executives feel this on a whole other level, specifically in terms of monthly development payroll—as seen in the next graph.

Cost/Revision



Release 1 was delivered with a monthly payroll of a few hundred thousand dollars. The second release cost a few hundred thousand more. By the eighth release, the monthly payroll was $20 million and climbing.

Compare the curve here with the lines of code written per release in our second graph. The initial few hundred thousand dollars per month bought a lot of functionality, but the final $20 million bought almost nothing.

Continuous Optimization

This ebook focuses on 4 core aspects of evaluating project health: Technology, Architecture, Setup and Tech Debt.

It's important to not only make your project great but keep it great as you move forward. And although the principles of Continuous Optimization can be applied at any stage of a project, it's easiest to incorporate it into the development lifecycle during project planning. By having carefully set up processes from the start, we're able to resolve problems before they become a significant threat to your business.

TECHNOLOGY

The main blockers—and primary focus points —of Continuous Optimization are **tight coupling** and **untested code.**

Introduction

Choosing the right technology is a complex task in theory, yet in practice it boils down to simply analyzing what's out there and how it fits your agenda.

One framework could seem absolutely perfect for your new project, but if you don't have engineers with enough know-how, you might be better off with the tech stack you already have expertise in. Or perhaps you're looking for, say, an emailing service—in which case, just pick your favorite and move on. No need for careful, weeks-long deliberation. Most of the time, you don't actually need perfect. You need good enough.

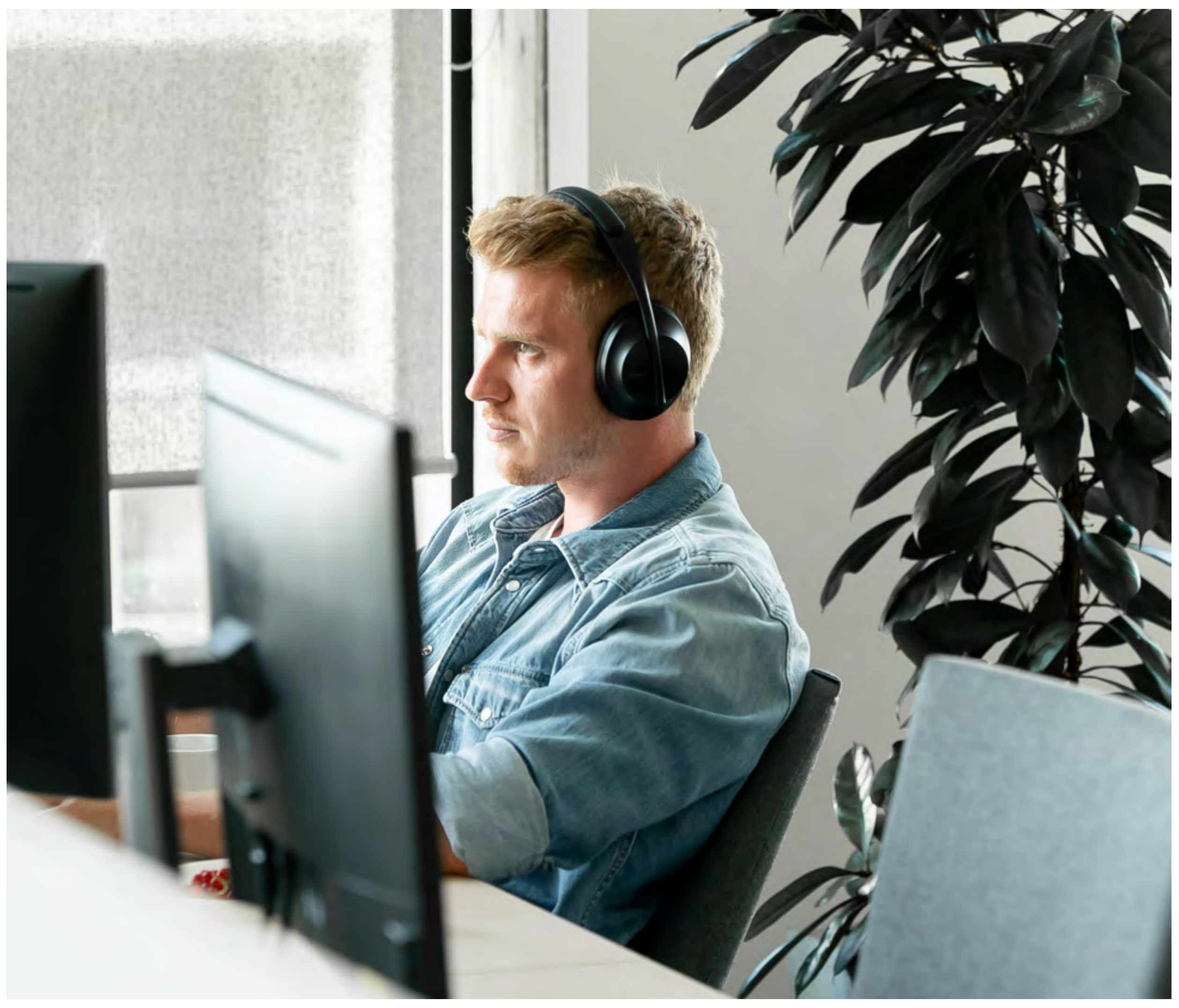
Sound like we're setting the bar low? Not at all. The reason that we don't want to make a specific technology the core of our business is because technologies should function solely as tools that help us achieve our goals. They should be easily replaceable once we get familiar with newer, better tools.

This doesn't only apply to third-party software. What actually changes more often is the functionality within our own codebase because business needs incessantly evolve. Which is why we need to structure the code in such a way that it's easy to move things around and recreate whole modules from scratch if needed.

The main blockers in doing this effectively are tight coupling and untested code—which are the primary focus points of Continuous Optimization.

Legacy Code

Legacy code is any code you are afraid to change. That fear could stem from insufficient test coverage, outdated technology or a complex dependency relationship between your modules. Changing it poses a high risk because it can introduce unforeseen regression in all code that depends on it. Oftentimes, this code is maintained by a handful of individuals who "know how it works" and any attempts to introduce new engineers into this circle are accompanied by frustration and high business risks.



Symptoms

- Insufficient test coverage (or no tests at all)
- Only a small subset of engineers knows how to maintain the code
- The technology used is outdated

Business Risks

- Adding new features or changing existing features becomes slower
- Domain knowledge gets concentrated into a small group of engineers, increasing the bus factor
- Changes have a high risk of causing regression across multiple projects that depend on the legacy code

49%

49% of engineers view tech debt/legacy code as **the biggest technical barrier** for the companies at which they work.

STRV Tip

Adding unit and acceptance tests allows you to stabilize the existing behavior, which lets you make your changes with greater confidence. Creating a solid test setup is the first step towards tackling legacy code effectively.

Outdated Technology

Outdated technology is any functionality that could be replaced by its modern counterpart. Keeping outdated tech in the codebase leads to sluggish development, security vulnerabilities and overall higher cost per feature. Using these tools oftentimes hampers the engineers' experience, who in turn become frustrated with slower progress. Onboarding new people becomes harder as well because junior engineers may not be familiar with the technology used and, if it is niche enough, there may not even be enough professionals on the market.

- Outdated technology usually has different priorities or was developed with different use cases in mind
- Older technologies tend to increase latency for users
- Finding sufficient expertise for code maintenance becomes increasingly more expensive
- Onboarding takes longer and processes take more time with tools which are slower compared to their modern counterparts

Symptoms

- External dependencies are not actively maintained (no LTS)
- It is difficult to find people with expertise in the given technology
- The engineer experience suffers, as tools don't produce fast and quality outputs

Business Risks

- Older technologies can approach problems in ways that are incompatible with modern design principles, forcing engineers into outdated processes

External Dependencies

Externalizing functionality opens up windows for potential safety breaches and makes code harder to maintain. Reducing such dependencies brings more control over code quality in the long term.

Symptoms

- Bundle size is bigger than necessary
- Heavy reliance on third-party services

Business Risks

- Fixing external dependencies can prove troublesome, as most of them are a "black box"
- Security issues
- Stability problems
- Slower migration to different services

STRV Tip

Isolating external dependencies behind an API layer gives you greater control over changes. Regularly keep dependencies up-to-date and find alternatives for the dependencies which are no longer maintained, or bring the functionality in-house if possible.

50%

Approximately 50% of the time engineers spend on **maintaining code** is actually spent trying to understand the code that they are working to maintain.

ARCHI-TECTURE

As a project evolves and adapts to new business needs, the architecture must continue **leading the way.**

Introduction

Architecture is a crucial focus point during a project's planning stages. When you think of architecture in terms of buildings, you imagine it as something that doesn't change once construction commences. But when it comes to engineering, it's a whole other story.

The main difference is the software development lifecycle.

As a project evolves and adapts to new business needs, the architecture must follow—and not only follow, but lead the way because all changes need to be reflected in the updated design. You wouldn't install a balcony to a window without adding it to the blueprints first.

The person or team in charge of the architecture should partake in the development process to better understand project needs. Only then can it be augmented effectively, as "ivory tower" design often omits important changes to existing concepts.

Layering the functionality with clear boundaries focused on core business entities and use cases provides a clean form of a plugin-based system that can be enhanced or replaced easily while limiting unwanted regression across modules.

Bad Project Architecture

Good architecture is one that not only works but is also scalable and maintainable. It shouldn't rely on tight coupling with external services, and it should clearly define functional layers.

Bad project architecture makes onboarding difficult and creates a brittle dependency tree. More importantly, small changes can cause rippling regression throughout the whole project.

Symptoms

- Confusing structures make it difficult to decide where a feature should go
- Lack of interface boundaries between layers
- Circular dependencies
- Small changes affect multiple parts in the codebase
- Tight coupling between the business layer and external dependencies

Business Risks

- Onboarding new people takes longer
- Functionality has no defined ownership of behavior, and changes in dependencies directly affect the functionality and propagate further in the dependency tree
- Difficulties with migration to different services or scaling

90%

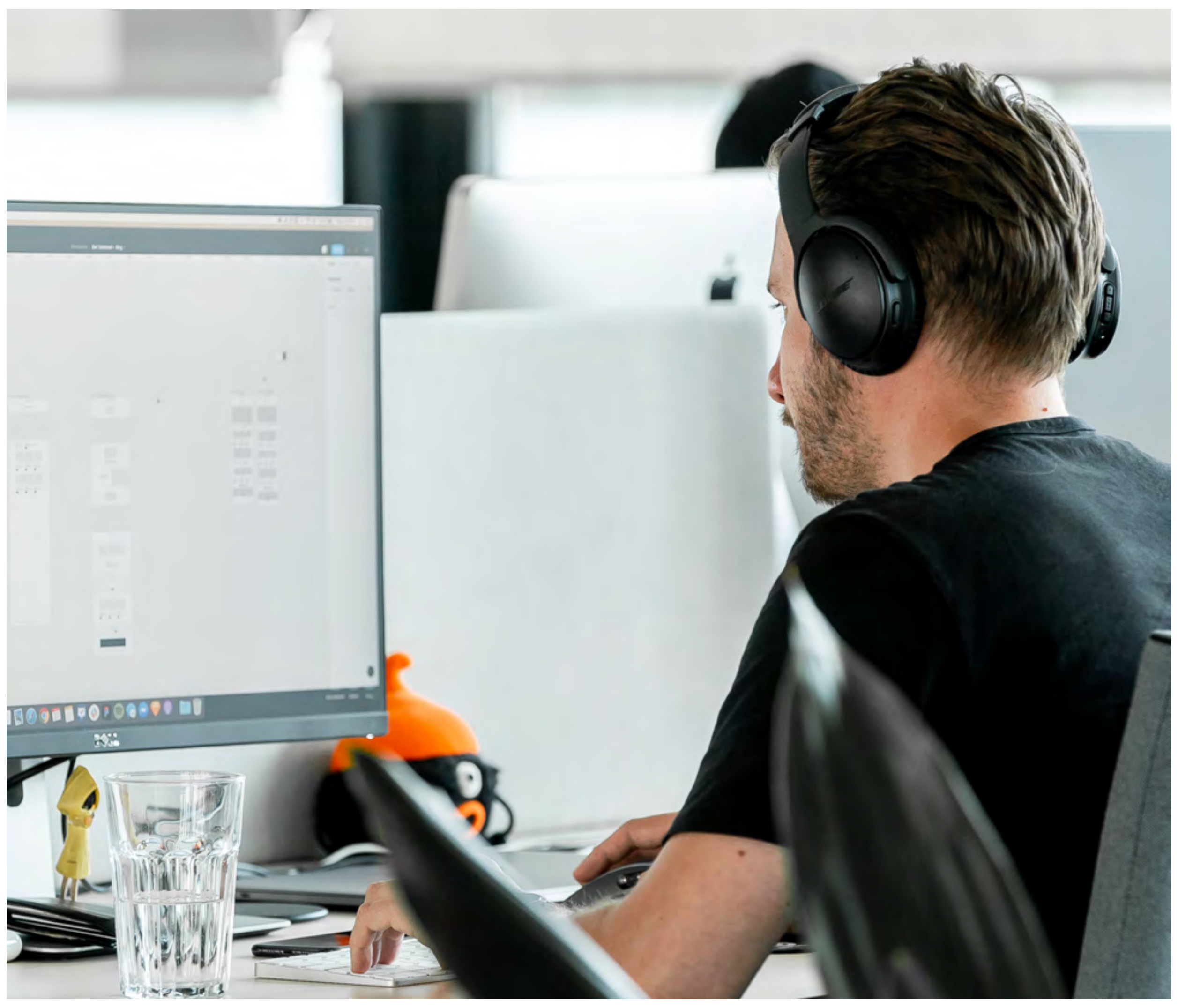
By 2022, 90% of all new apps will feature **some form of composable architecture that improves** the ability to design, debug, update and leverage third-party code.

STRV Tip

Introduce non-circular API layers across modules for greater control and ownership of responsibility. Try to move external dependencies to the outermost layers; by doing so, you facilitate an easier plugin-based system where you can easily swap services.

Resource Sharing

Code reuse is an important thing to get right in a project to prevent duplicity issues down the road. A good setup allows multiple codebases to effectively share functionality across projects, and it reduces the chance that updating one resource will negatively affect another. As shared resources are inherently coupled, it's important to strike the proper balance between ease of use and modularity.



Symptoms

- Duplicate business logic across projects
- Increased bundle size
- Rigid external dependencies require workarounds to implement properly

Business Risks

- Updating business logic becomes error-prone or inefficient
- Duplicate implementations tend to increase latency for users
- Codebases become larger and harder to maintain, increasing the cost-per-line of code

STRV Tip

Decouple frequently used functionality into a separate module, ideally based on business logic. However, make sure there is clear ownership of the module and that it can be iterated on efficiently as it becomes a tightly coupled dependency.

90%

In 2015, Hitachi Consulting commissioned a study that found **legacy systems** were holding back 90% of businesses.

SETUP

Up-to-date documentation greatly simplifies the workflow, so introducing **helpful systems** becomes invaluable as a project grows.

Introduction

Project setup is one of the most important aspects of any project. A high-quality setup could relieve many headaches down the road. Due to this, it's best to use common design patterns because they simplify onboarding onto the project.

It's crucial to include code quality tools to prevent code decay and to maintain the high integrity of code style throughout the codebase. These checks can be easily automated in the CI/CD pipeline, along with other tools used to optimize the condition of the project.

Additionally, it's best to remove as much of the human element as possible from repeating processes in order to avoid oversight mistakes. Great contenders for this are infrastructure—ideally defined in the codebase via configuration files—and deployment.

An oftentimes overlooked aspect assumed to be of lesser importance is documentation. Up-to-date documentation greatly simplifies the workflow for anyone who needs to interact with the project as a whole or with its various parts, so introducing helpful systems becomes invaluable as a project grows.

Keep in mind that documentation isn't only readmes, but architectural diagrams, comments or version system commits, all of which should conform to the agreed-upon format.

CI/CD Setup

Continuous Integration (CI) and Continuous Deployment (CD) should be a part of every production-grade project. Their primary purpose is to delegate various checks to automated services instead of relying on human checks.

Symptoms

- Inconsistent codebase style
- Reappearing bugs
- Deployment failures
- Recurring "broken pipeline" blockers

Business Risks

- A lot of time spent on maintenance and infrastructure bug fixing
- An increased amount of bugs
- Slower deployment cycles

70%

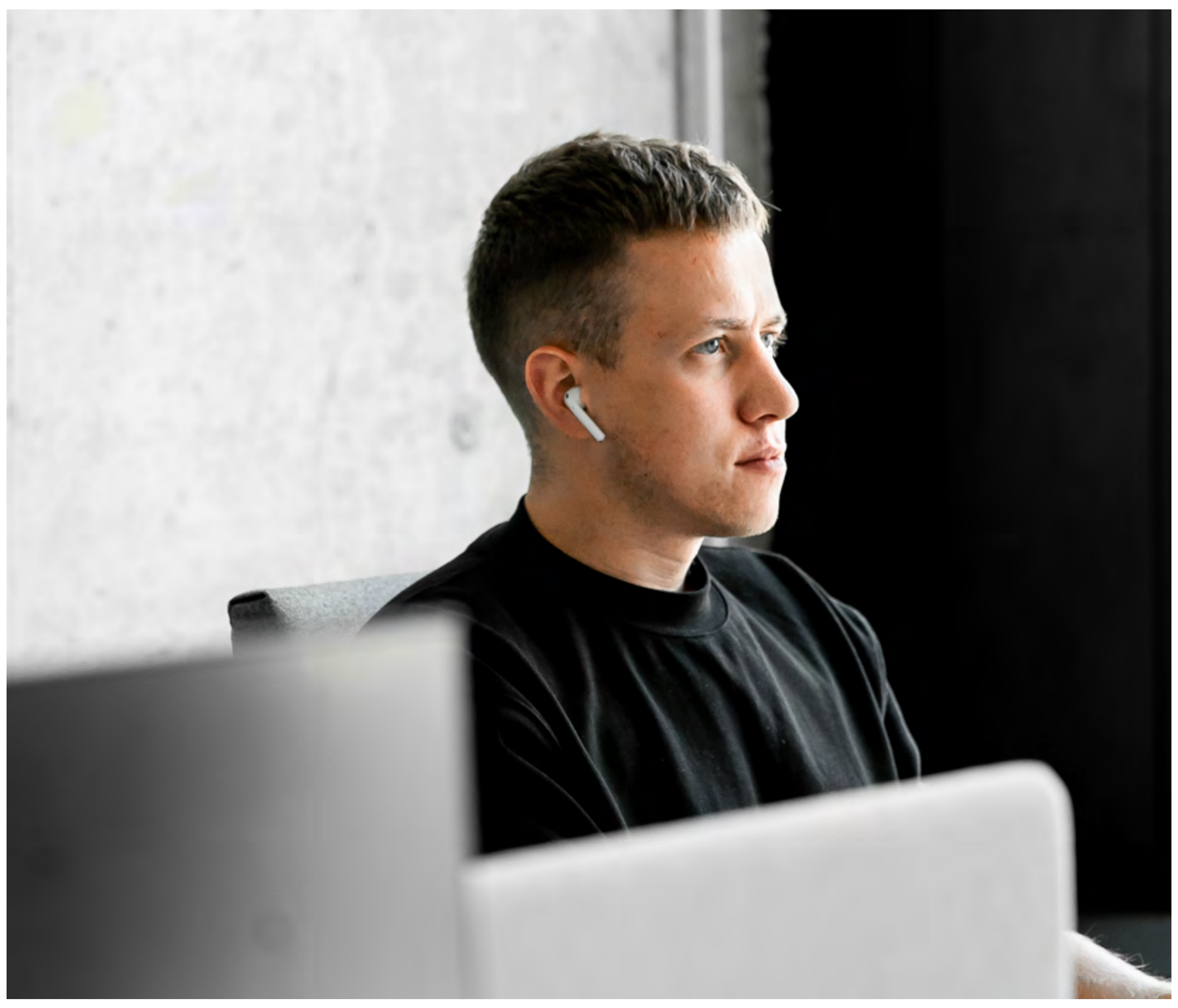
Maintenance consumes over 70% of the total life-cycle cost of a software project.

STRV Tip

A healthy CI/CD setup should not only handle automated deployments but should also guarantee the codebase quality, ensuring that the code is properly formatted and tested before reaching production.

Documentation

Documentation helps with onboarding and increases the usability factor of any project. Keeping it in a well-maintained condition can reap considerable benefits. The most relevant docs include project readme, architecture diagrams, usage of third-party services and public-facing API documentation.



Symptoms

- Architectural patterns are not clearly defined
- Difficulty finding appropriate resources
- If not properly maintained, can falsely describe the actual implementation
- Difficulty tracking changes

Business Risks

- Onboarding new people takes longer
- Increased tech debt
- If the product is public-facing API, it critically downgrades the adoption rate

STRV Tip

Document new features by including either a readme, comments or through a separate service. Implement mechanisms that ensure fresh state of the documentation, like properly formatted commits which can be compiled into a rigorous changelog and allow automated versioning of the system.

1 NO.

Effective CI/CD setup is the No. 1 way of eliminating the human factor in faulty deployments.

TECH DEBT

Tech debt is generally divided into two primary categories—**deliberate** and **accidental**—and stems from sacrificing quality over speed.

Introduction

Tech debt comes from sacrificing quality over speed. Like financial debt, it can be useful in the short term but can easily rack up interest if not properly maintained. Harder code maintenance, slow iteration times and reappearing bugs are just a few critical issues that stem from an inefficient tech debt mitigation strategy.

Tech debt can arise for multiple reasons, but we generally classify it using two primary categories: deliberate and accidental.

Deliberate tech debt happens when engineers sacrifice quality over speed. It’s fine in small amounts but should be recorded into the backlog and dealt with ASAP.

Accidental tech debt is acquired over time. It can be somewhat reduced by diligently upholding the code quality and architectural consistency; however, some tech debt will still accumulate. It’s crucial to allocate time in the backlog to identify and deal with it properly.

80%

The cost of buggy software in the U.S. totaled **$2.26 trillion in 2018.** That number did not include the cost of future tech debt. This data indicates why software developers spend 42% of their time fixing bugs, and why 80% of IT budgets go toward doing so.

STRV Tip

As Tech Debt is something for which we can plan, it's important to devise an early mitigation strategy based on expected project velocity. One of the most popular approaches is dedicating a sprint to fix identified issues. A second option is including a task focused on resolving the debt in every sprint.

Make sure the tasks in the backlog don't get buried by an onslaught of feature-focused tasks. Ensure that the team sees them as a priority, as it's the engineers who know the exact level of tech debt.

Final Note

We've said every project includes mistakes, and it's true. Being smart doesn't mean fighting for a miracle by aiming for zero mistakes. Being smart means preventing as many issues as you can and fixing what's already been busted before it leaves a mark.

Continuous Optimization is now less of a choice and more of a necessity. In the ever-evolving world of software development, it's all about having the right processes, the right realistic outlook and, yes, the right people on the job.

The STRV team is here for you if you want to discuss how we can help out. The foundation of this ebook came from our senior engineers, who wrote it simply because they wanted to share with others what they themselves took years to learn. So if you're looking for know-how and genuine enthusiasm, *that* we can guarantee.

Just reach out.

THIS EBOOK WAS LAST UPDATED ON AUGUST 10TH 2020.

AUTHOR: PETER VARHOLAK
EDITOR-IN-CHIEF: LINDA KRESTANOVA
DESIGN: LENKA SIVA