Continuous Modernization

The never-ending discipline of improving microservices, monoliths, distributed monoliths, individuals, and teams at scale

Diego Pacheco
Sam Sgro

www.bpbonline.com

First Edition 2025

Copyright © BPB Publications, India

ISBN: 978-93-65893-106

About the Authors

- **Diego, Co-Author**

  Diego Pacheco is a seasoned, experienced Brazilian software architect, author, speaker, technology mentor, and DevOps practitioner with more than 20+ years of solid experience. I've been building teams and mentoring people for over a decade, teaching soft skills and technology daily. Selling projects, hiring, building solutions, running coding dojos, long retrospectives, weekly 1:1s, design sessions, code reviews, and my favorite debate club: architects community of practices and development groups for more than a decade. Live, breathe, and practice real agile since 2005, coaching teams to help many companies discover better ways to work using Lean, Agile principles, and methods like XP and DTA. I've led complex architecture teams and engineering teams at scale guided by SOA principles, using a variety of open-source languages like Java, Scala, Rust, Go, Python, Groovy, Javascript and Typescript, cloud providers like AWS Cloud and Google GCP, amazing solutions like Akka, ActiveMQ, Netty, Tomcat and Gatling, NoSQL databases like Cassandra, Redis, Elasticache Redis, Ekasticsearch, Opensearch, RabbitMQ, libraries like Spring, Hibernate, and Spring Boot and also the NetflixOSS Stack: Simian Army, RxJava, Karyon, Dynomite, Eureka, and Ribbon. I've implemented complex security solutions at scale using AWS KMS, S3, Containers (ECS and EKs), Terraform, and Jenkins. Over a decade of experience as a consultant, coding, designing, and training people at big customers in Brazil, London, Barcelona, India, and the USA(Silicon Valley and Midwest). I have a passion for functional programming and distributed systems, NoSQL Databases, an obsession for Observability, and always learning new programming languages.

  Currently working as a principal Software Architect with AWS public cloud, Kubernetes/EKS, performing complex cloud migrations, library migrations, server and persistence migrations, and security at scale with multi-level envelope encryption solutions using KMS and S3. While still hiring, teaching, mentoring, and growing engineers and architects. During my free time, I love playing with my daughter, playing guitar, gaming, coding pocs, and blogging. Active blogger blog at diego-pacheco.blogspot.com.

- **Sam, Co-Author**

  Sam has always been passionate about applying engineering to complex business domains. Sam started his career in open-source cryptography before joining an early-stage bioinformatics & data analytics startup with a successful exit to Thomson Reuters. Sam then worked as an engineering leader, leading teams on multimillion-dollar growth initiatives in pharmaceutical research and academic literature analysis. In 2018, Sam switched gears to FinTech by joining a FinTech company in the Bay Area as its Chief Architect, where he now serves as Head of Consumer Engineering, driving its digital banking initiatives.

About the Reviewer

Garen Mnatsakanov is currently leading Engineering at a FinTech company in the Bay Area. After college, he took on every opportunity in tech that came his way and this has led him to gain a ton of invaluable experience. To this day, he tries to stay hands-on and code, even though being in a leadership role it is harder to find the time. Garen thinks it is a must if you want to stay technical, especially in the ever-changing field of software engineering. He has a passion for building effective product engineering teams that can work closely together to deliver great products.

To Diego and Sam,

I want to thank you again for the opportunity to be the technical reviewer of your second book. Once again, you guys have demonstrated your passion for software architecture and modernization and towards building a strong engineering culture. A culture that enables businesses to grow, succeed, and scale, leveraging their technology instead of being burdened by it.

To my mom, dad, sister, and nephew, thank you for your love and care. And to my wife Andrea, thank you for your support and encouragement, it means a lot to me.

-Garen Mnatsakanov, Technical Reviewer

Acknowledgements

- **Diego, Co-Author**

  Thanks, God, Thanks, God, Thanks, God. I appreciate all my blessings, I wrote this book with love, passion, and lots of hard work. I wish we could share the same passion for software architecture, design, and complex problems. Deeply rooted in this, you can make a big impact in your organization and grow in your career and as a human being. Thank you for buying my book, I really appreciate it. I hope my experience and perspectives guide you in your journey. No matter if you are a software architect, software engineer, engineering manager, DevOps engineer, QA engineer, frontend engineer, director, VP, or CTO.

  I have a deep passion for technology, especially for software architecture. My passion could only happen due to the immense support of my loved family, my wife Andressa, and my dear daughter Clara. My dear friends Margarida, Adao, Israel and Tais, Jun, Richard, Ty, and many other friends are not named here, but be sure you have a place in my heart… Brazil!

  A small disclaimer: This book does not reflect the ideas, decisions, or opinions of any of my past or future employers or customers in my last 20+ years of experience with distributed systems and systems at scale, working for companies, and doing consultancy.

- **Sam, Co-Author**

  The ideas in this book are the continuation of the work we started in *Principles of Software Architecture Modernization*. I am grateful to get the opportunity to continue the themes we started in that book, to help educate teams on what it really means to change code and change their companies. In turn, I hope it inspires you and propels you forward on your journey of professional transformation, to assemble great teams and work on amazing things.

  To my friends and family across Canada, the US, Spain, the UK, and Brazil, and especially my beloved wife Claudia and children Kat and Erica, thank you for your patience and for giving me the space to do all the work needed to see these ideas hit print.

Preface

Why did we write this book?

After writing our first book, *Principles of Software Architecture Modernization*, we realized we had more things to say: about culture, about discipline, feedback, and principles to help software engineering teams realize they CAN change their software and don't have to stay stuck in the past.

Our first book heavily discussed the problems of working in legacy software, but you do not always have a budget or modernization project. We wanted to review all the invisible things, new ways of working, and the secret power of engineers who hold the power in their own hands. With discipline and principles, anything can change.

We also realized that selling a modernization project is a big deal, and takes time, preparation, and patience even beyond the assessments we wrote in *Chapter 5*, *Assessments* of *Principles of Software Architecture Modernization*. A background in sales and negotiation is useful for everyone, even and especially the engineers on the ground.

We wanted to challenge the current ways of working in big companies. Process is not always the answer to every problem. Instead, the root of real change lies in company culture. If they don't believe in modernizing technology, don't change the goal - change the company itself.

Our book will not give you easy answers or a magic formula for success. We are here to make you think and explore different options to change the way you think and book. Continuous Modernization is a philosophy book, even more so than our first. If you want a magic wand to fix your problems, this book is not for you.

We will be very visual in this book, so expect a lot of diagrams to help convey our points. Our book is Java-centric but not code-heavy; you will see some pseudocode examples but do not expect complete applications built end to end. This is not a tutorial book.

We will be very technical in this book and will connect many different subjects, but don't worry; we will explain things in depth and with lots of practical scenarios and examples. The topics we will cover are wide-ranging, and sometimes we will review the same points from different angles to uncover different perspectives. We hope you like it, and thank you in advance for your readership.

However, while reading this book, you can expect:

- Examples: Practical examples from our experience in technology.
- Tradeoff Analyses: Architecture is all about tradeoffs, so expect many comparisons of pros and cons.
- Figures: Many diagrams and pictures to illustrate scenarios, tradeoffs, and options.
- Multiple Options: We will provide multiple options and the best analysis to make you consider the entire problem space.
- Repetition: We will repeat some principles over and over, analyzing them in different contexts to gain new understanding.
- Summary & Learning: Every chapter will have a summary of things to remember from each chapter. This is a long book, and you might need to read it multiple times; make notes in whatever way works for you, and look back on what you find interesting or disagree with. (The authors love disagreements!) Remember that when you just passively read (input) you don't learn as much as when you produce (output). You can write a blog post, run a lightning talk or presentation to your engineers or company, talk to a friend, it is important to produce output, and we believe that's the best way to learn anything, not only this book.

  In summary, this book will not have:

- Easy Answers: There is no magic formula to fix your monoliths, just options we will help you navigate and digest to find your own answers. No quick fixes to your complex problems, just reality.
- Tutorials: You won't find step-by-step instructions on how to build applications. This is not a tutorial book.

Book Structure

The book is organized in the following chapters:

Chapter 1 - What is Continuous Modernization?

Chapter 2 - Unlearning

Chapter 3 - Discipline & Feedback

Chapter 4 - Decisions & Tradeoffs

Chapter 5 - Stability & Troubleshooting

Chapter 6 - Opportunistic Design

Chapter 7 - Continuous Refactoring

Chapter 8 - The Art of Selling

Chapter 9 - Effective Negotiation

Chapter 10 - Culture Shift

Chapter 11 – Epilogue

Our chapters are divided into three important sections: Learning, Execution, and Scaling. First, you need to learn new skills that will change how you approach problems. Execution is about getting things done. Scaling is how we can have a bigger impact on the company, from bigger modernization projects to changing the company itself. Let's dive a bit more into each chapter.

Learning

Understanding what continuous modernization is all about. Shifting your mindset and upskilling yourself with powerful and game-changing soft skills to make better software every day, day-by-day. First you improve yourself, then you improve the software you build.

Covered in chapters:

Chapter 1 - What is Continuous Modernization?

Chapter 2 - Unlearning

Chapter 3 - Discipline & Feedback

Chapter 4 - Decisions & Tradeoffs

Execution

Approach execution through a different lens and apply improvements even without a modernization project. Execution is all about practical, day-to-day scenarios that will change how you and your team behave and achieve better results.

Covered in chapters:

Chapter 5 - Stability & Troubleshooting

Chapter 6 - Opportunistic Design

Chapter 7 - Continuous Refactoring

Scaling

After establishing a successful track record with lots of quick wins, now we can take things to the next level. Selling modernization projects takes both technical skills and soft skills. Negotiating a good outcome means understanding how to speak to management without compromising your values. Culture is the final barrier to any modernization initiative, and Continuous Modernization will show you how to improve not just your software and teams but the company itself.

Covered in chapters:

Chapter 8 - The Art of Selling

Chapter 9 - Effective Negotiation

Chapter 10 - Culture Shift

Chapter 11 - Epilogue

Coloured Images

Please follow the link to download the
Coloured Images of the book:

https://rebrand.ly/5b2uqrq

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1. What is Continuous Modernization? 1

Structure 2

Legacy Systems 3

What is a legacy system? *4*

Legacy systems deliver value *4*

Working with legacy systems *5*

Amplifiers *5*

Backfire *7*

Opportunities *7*

Waste, Technical Debt, and Anti-Patterns *9*

Modernization Projects *10*

What if you don't have a modernization project? No Budget? *10*

What if your people are not convinced? *11*

What if you can't re-write it all? *12*

Continuous Modernization is a Philosophy 13

The Snowball Effect: Small improvements lead to big outcomes over time *14*

Learning and Upskilling *16*

The True Nature of Software Development *16*

Growth vs. Fixed Mindset *18*

Behavior Iceberg *18*

Limiting Beliefs *19*

What should we learn first? *19*

You are the product *22*

Continuous Modernization Execution 23

Consultant Mindset *25*

Strategy and Techniques *26*

Stability & Troubleshooting *26*

Opportunistic Design *26*

Continuous Refactoring *26*

Kup książkę

Scaling ... 26

Going beyond process ... 27

The Art of Selling ... 27

Effective Negotiation ... 27

Culture Shift ... 27

Common Mistakes ... 27

#1 Anti-pattern: Modernization is just for big projects! ... 28

#2 Anti-Pattern: My team needs permission! ... 28

#3 Anti-pattern: We already changed too much, we are good! ... 28

#4 Anti-pattern: Modernization does not work for us, we are unique ... 29

#5 Anti-pattern: No time, No people to do anything ... 29

Things to Remember ... 29

2. Unlearning ... 33

Structure ... 33

Unlearning #1: Code is a liability ... 35

Unlearning #2: Timelines above it all ... 36

Unlearning #3 - You cannot modernize and deliver ... 37

Unlearning #4 - User experience is just look-and-feel ... 38

Customer Experience (CX) ... 40

Unlearning #5 - Only juniors need to learn ... 40

Unlearning #6 - Don't touch it, do the minimal ... 42

Unlearning #7 - I can't convince my team ... 43

Being the last vs. being the first ... 44

Learning > Delivery ... 45

Shallow vs. Deep Mindset ... 45

Shallow vs. Deep: Dealing with Bugs ... 47

Shallow vs. Deep: Dealing with Features ... 48

Shallow vs. Deep: Code Migrations ... 49

Shallow vs. Deep: Learning ... 50

Beyond learning on demand! Effective Learning ... 52

Input ... 53

Reading ... 53

Output 53
Taking more from Feedbacks 55
The Cost of Silence 56
Learning from experiences and failures 57
The issue with Delta reviews 58
Discovery with Coding POCs 59
Creating the environment you want to work in 59
Hiring & Growing Talent 60
Golden Goose Mentality (turning bad systems into good projects) 60
How do you manage your day - your agenda decides what you deliver or not 61
Common Mistakes 62
#1 Anti-Pattern: Analysis Paralysis 62
#2 Anti-Pattern: Will impact timelines 62
#3 Anti-pattern: I have no questions 63
#4 Anti-pattern: I have no time for POCs 63
#5 Anti-Pattern: I have no control over my agenda 63
Things to Remember 63

3. Discipline & Feedback 67
Structure 67
Discipline 69
Limits of Compliance & Standards 70
What discipline really is about 70
Interruptions 73
Triage 75
Eisenhower Method 76
Deep Work 76
Playing with discipline 76
Agenda 77
More time or more focus? 78
XP Baby Steps 79
Crawl, Walk, Run, Fly 79
Experiments 79

Daily ... 80
Feedback ... 80
Feedback usually sucks ... 81
Feedback common mistakes ... 82
Good vs. Bad Feedback ... 82
The Cost of Firing People ... 85
The Cost of NOT Firing People ... 86
Economics of Feedback ... 86
Limits of Group Meetings ... 89
Retrospectives ... 90
Embedding Feedback ... 92
1:1s ... 92
Common Mistakes ... 93
#1 Anti-Pattern: Standardization over discipline ... 93
#2 Anti-Pattern: Waterfall ... 94
#3 Anti-Pattern: No Feedback, Shallow Feedback ... 94
#4 Anti-Pattern: Too many meetings ... 94
#5 Anti-Pattern: No Retrospectives ... 94
#6 Anti-Pattern: 15 min Retrospectives ... 94
#7 Anti-Pattern: No 1:1s ... 94
Things to Remember ... 95

4. Decisions & Tradeoffs ... 99
Structure ... 100
Thinking vs. Acting ... 101
A bias to action should not be your only bias ... 101
Meeting-first vs. Writing-first cultures ... 103
The power of thinking clearly ... 105
Avoiding Analysis Paralysis ... 107
Strength in Depth vs. a Shallow Understanding ... 108
Tradeoffs ... 109
Choice is Everything ... 109
The Art of Tradeoffs ... 110

Sidebar: Laying Strong Foundations ... 112
Tradeoff Analysis In Practice ... 112
Tradeoff Pyramid ... 112
Continuous Modernization Tradeoff Analysis ... 114
Understanding the Requirement, or "Why?" ... 114
Anti-pattern: Order-taker ... 115
Anti-pattern: Requirements contain the solution ... 116
Sidebar: The Five Whys ... 116
Identify Options, or "How" ... 117
Inventory Advantages and Limitations ... 120
Knowing your limits ... 120
Playing to your Advantages ... 121
Analyze Tradeoffs ... 122
Scope, Quality, Time, and Cost ... 122
One-way vs. Two-way doors ... 124
Modernization vs. Debt ... 125
Time vs. Flexibility ... 126
Co-dependence vs. Isolation ... 126
Buy vs. Build ... 127
Size vs. Velocity ... 127
Time vs. Discovery ... 128
Sidebar: Research and Understanding ... 129
Techniques to discover and model tradeoffs ... 131
Decide ... 134
Communicate and Commemorate: Decision Logs ... 135
Every decision is eventually wrong ... 136
Strategy vs. Tactics ... 136
Opportunistic Design and Continuous Refactoring ... 137
Common Mistakes ... 137
#1 Anti-pattern: Pressure to Deliver ... 137
#2 Anti-pattern: Accepting requirements at face value ... 138
#3 Anti-pattern: False confidence, shallow knowledge ... 138
#4 Anti-pattern: Reaching conclusions before the decision ... 138

#5 Anti-pattern: Only listening to the loudest voices 138
#6 Anti-pattern: Letting emotion overrule logic 138
#7 Anti-pattern: Customer comes last 139
Things To Remember 139

5. Stability & Troubleshooting 143
Structure 144
Importance of Stability 145
Reliability vs. Stability: how to sleep at night 148
Progress on bad code begins with good foundations 150
Introducing the Stability Mindset 151
Stability and Troubleshooting in Practice 153
Observability: Insight is everything 154
Do you know what is happening? 154
Logs 154
Distributed Tracing 156
Metrics 157
Dashboards 160
Alerts 162
Signal vs. Noise 165
Troubleshooting 167
Proactively Investigating Code For Failure 167
The Importance of the Scientific Method 169
Step 1 – Observe 170
Step 2 – Research 170
Step 3 – Create a Theory / Hypothesis 171
Step 4 - Experiment 171
Step 5 – Analyze Data 172
Step 6 – Report Conclusions 173
Step ++ - Iterate 173
Sidebar: debuggers and profilers 174
Introducing Tests 177
Common Mistakes 178

#1 Anti-pattern: Thinking Stability means stasis *179*

#2 Anti-pattern: Believing Stability is the same as Reliability *179*

#3 Anti-pattern: Throwing logs at the problem and calling it done *179*

#4 Anti-pattern: Creating way too many alerts *179*

#5 Anti-pattern: Failing to diagnose the underlying problems driving your problem *179*

#6 Anti-pattern: Allowing management to override science *180*

#7 Anti-pattern: Ignoring tests *180*

Things to Remember 180

6. Opportunistic Design 183

Structure 183

Bad Designs 184

Modern Designs 186

What is design about? *186*

Elements of Good Designs *190*

Design multiple solutions *192*

The Issue with Designs and Abstractions *198*

See the invisible *198*

Planetary Alignment (win-win) 202

Perception that Features always come before improvements *203*

Features are the best opportunity for change; just do it differently *204*

Opportunistic Design 204

2 steps forward, 1 backward *204*

"Slipstream" technique *207*

Common Mistakes 208

#1 Anti-Pattern: There is no time, we need to deliver *208*

#2 Anti-Pattern: I don't have a modernization project *209*

#3 Anti-Pattern: The problem is too big, we need a complete re-write *210*

#4 Anti-Pattern: No one asked me to do refactoring *210*

#5 Anti-Pattern: Dropping the Ball *210*

#6 Anti-Pattern: Refactoring the style of the code *210*

#7 Anti-Pattern: A lot of things can go wrong, let's stick to the standard *211*

Things to Remember 211

7. Continuous Refactoring ... 213

Structure ... 213

Mindset ... 214

Campground Rules ... 214

Leveraging Open Source ... 219

Daily Migrations (small bumps avoid big bumps) ... 230

Talking through Code ... 230

Dealing with dependencies via Contract Mocking ... 233

Deleting Code ... 236

Directly Referenced ... 238

Private Visibility ... 238

Default Package / Protected Visibility ... 239

Public Visibility ... 239

Indirectly Referenced ... 240

Workspace Usage ... 240

Repository Search ... 240

Observability ... 242

Dealing with testing ... 243

Common Mistakes ... 243

#1 Anti-Pattern: I have no time for refactoring ... 244

#2 Anti-Pattern: There are no tests; we need a test env to test ... 244

#3 Anti-Pattern: Delete and see what happens in production ... 244

#4 Anti-Pattern: My team does not want refactorings or testing ... 245

#5 Anti-Pattern: It's impossible to refactor the 3K LoC class! ... 245

#6 Anti-Pattern: Let's remove all boilerplate code and have it in a wrapper library ... 245

#7 Anti-Pattern: We need to deliver features; why do we need to do migrations ... 246

Things to Remember ... 246

8. The Art of Selling ... 249

Structure ... 250

Why sales is important to engineering ... 251

A big leap forward ... 252

Timing is everything ... 255

Building on Your Success ... 256
People are the problem, and the solution ... 257
Selling Your Idea ... 258
Identifying Key Decision Makers ... 259
Selling your decision-makers ... 262
Tapping into emotion to drive modernization ... 262
Key Selling Points ... 265
Communicating Effectively ... 267
Managing Upwards ... 267
Managing Downward ... 269
Communication Techniques ... 270
Narrative: the steel thread ... 271
Written Communication ... 272
Verbal Communication & Presentations ... 274
Common Mistakes ... 278
#1 Anti-Pattern: Selling is distasteful, or something only managers do ... 278
#2 Anti-Pattern: Not taking credit for past successes ... 278
#3 Anti-Pattern: Choosing the wrong time or place ... 279
#4 Anti-Pattern: Only managing upwards ... 279
#5 Anti-Pattern: Only managing downwards ... 279
#6 Anti-Pattern: Lacking empathy for others ... 279
#7 Anti-Pattern: Failing to craft a narrative ... 279
Things To Remember ... 280

9. Effective Negotiation ... 283
Structure ... 284
You got them to the table. Now what? ... 285
Negotiating Complex Business Tradeoffs ... 287
The Power of Trust ... 287
Empathy and Active Listening ... 290
Sidebar: On Selling Purely Technical Projects ... 294
Aligning Objectives ... 295
When to Compromise ... 296

Selling establishes value, negotiation trades value 296
Compromise is a double edged sword 297
Time vs. Money 299
Time vs. Focus 300
One way vs. Two way doors 300
Team Location vs. Money 301
Team Capacity vs. Feature Pipeline 301
Flexibility vs. Principles 303
Finding Reasonable Compromises 305
Preparation is Key 305
Aligning Expensive vs. Cheap 306
Dealing with Politics 307
Arriving at a Plan 309
Always Mistrust Estimates 309
Planning Must Not Become Waterfall 310
Common Mistakes 312
#1 Anti-Pattern: Loss of Trust 312
#2 Anti-Pattern: Failing to Listen 312
#3 Anti-Pattern: Compromising immediately 312
#4 Anti-Pattern: Trying to please everyone 313
#5 Anti-Pattern: Compromising your Principles 313
#6 Anti-Pattern: Trusting Your Estimates 313
#7 Anti-Pattern: Pursuing Certainty 313
Things to Remember 314

10. Culture Shift 317
Structure 318
What's wrong with our process? 319
What's wrong with management? 321
What's wrong with work? 323
Culture Matters 325
Culture is expressed in every decision 325
Culture Eats Strategy for Breakfast 327

Cultural Revolutions in Tech 329

Agile *329*

Lean *331*

Facebook: Move Fast, Break Things *333*

Netflix: No Rules Rules *334*

Blameless Culture *335*

Advancing your organization's culture 337

The Right Way To Scale *337*

Leveling Up Our Culture *339*

Ideal cultural values *341*

How do we shift culture? 344

Continuous Modernization as Culture Shift *345*

Culture Shift In Practice *346*

Common Mistakes 348

#1 Anti-Pattern: Roadmaps Written In Stone *348*

#2 Anti-Pattern: Management Is About Control *349*

#3 Anti-Pattern: Ignoring Culture *349*

#4 Anti-Pattern: Dark Agile *349*

#5 Anti-Pattern: Dogma over Understanding *350*

#6 Anti-Pattern: Failing to Learn and Grow *350*

#7 Anti-Pattern: Thinking Change Happens Overnight *350*

Things to Remember 351

11. Epilogue 353

Structure 353

Never-ending War 354

The Dark Side of LLMs 354

Patience 356

Slow but Steady Change 357

Direction vs. Speed 358

Index 361-367

CHAPTER 1

What is Continuous Modernization?

Continuous improvement is better than delayed perfection.

- Mark Twain

Diego worked as a consultant with a financial service company in 2008. There was a meeting with the CEO to strategize a plan for a new system built from scratch in Java. Our company had a legacy system written in Clipper[1] that was slow and did not scale. The CEO said, "I want a system that can last many years. I don’t want to be re-writing systems all the time.” That was such a powerful statement. Diego was thinking: “How can one write a system that lasts many years?” After more than twenty years of working with software in several industries, we can spot some common themes, one being that systems are never good enough. Therefore the need to modernize.

Today’s environments did not get any better; instead, the systems are much more complex and entangled. Legacy systems, web, mobile, monoliths, distributed monoliths, and on-prem data centers are mixed with multiple cloud providers and multiple SaaS vendors. We could just give up due to all this complexity. Hopefully, there is another way we can survive and navigate such complex environments and do better. If you ask any engineer what they want to do, the answer is unanimous: refactoring. Engineers hope that if they perform refactoring, problems will go away. Often, this hope is bundled into a modernization project. But what if you don't have this project? What if you must wait 3-5 years for such a project? What do you do in between?

1. Clipper programing language https://en.wikipedia.org/wiki/Clipper_(programming_language)

Continuous Modernization is a mindset. It's a fresh take on day-to-day problems. Surviving, adding value, and making better products where before might look impossible. Continuous Modernization implies working with legacy systems, monoliths, and distributed monoliths. Continuous Modernization is how we will reduce your future problems and improve products while delivering features, refactoring, and improving code daily.

Structure

In this chapter, we will cover the following topics:

- Legacy Systems
    - What is a legacy system?
    - Legacy systems deliver value
    - Working with legacy systems
    - Amplifiers
    - Backfire
    - Opportunities
    - Waste, Technical Debt, and Anti-Patterns
- Modernization Projects
    - What if you don't have a modernization project? No Budget?
    - What if your people are not convinced?
    - What if you can't re-write it all?
- Continuous Modernization is a Philosophy
    - The Snowball Effect: Small improvements lead to big outcomes over time
    - Learning and Upskilling
    - The True Nature of Software Development
    - Growth vs. Fixed Mindset
    - Behavior Iceberg
    - Limiting Beliefs
    - What should we learn first?
    - You are the product
- Continuous Modernization Execution
    - Consultant Mindset
    - Strategy and Techniques

        - Stability and Troubleshooting
        - Opportunistic Design
        - Continuous Refactoring
- Scaling
    - Going beyond process
    - The Art of Selling
    - Effective Negotiation
    - Culture Shift
- Common Mistakes
    - #1 Anti-pattern: Modernization is just for big projects!
    - #2 Anti-Pattern: My team needs permission!
    - #3 Anti-pattern: We already changed too much, we are good!
    - #4 Anti-pattern: Modernization does not work for us, we are unique
    - #5 Anti-pattern: No time, No people to do anything
- Things to Remember

Legacy Systems

All companies have legacy systems, usually more than one. Legacy systems are often very bad, full of anti-patterns, technical debt[2], bad decisions, bad code, and lack of tests. Engineers don't like working with legacy systems, and there is constant tension, frequently leading to a high turnover (*Figure 1.1*).

***Figure 1.1**: Legacy System - 2011 US Navy MS-DOS food service management system*

Source: https://en.wikipedia.org/wiki/File:US_Navy_110129-N-7676W-152_Culinary_Specialist_3rd_Class_John_Smith_uses_the_existing_DOS-based_food_service_management_system_aboard_the_aircraft.jpg

2. Technical debt by Fowler https://martinfowler.com/bliki/TechnicalDebt.html

What is a legacy system?

A legacy[3] system is a computer system that is outdated but still being used. Most of the time, legacy systems still exist due to the cost of refactoring and the cost of migrating to new systems. Such systems often have:

- **Outdated Technology**: Technology that is not the most optimal way of doing things. For instance, we could have a user interface in Java using JSF. However, currently, the common way to do it would be using NodeJS and React or meta-framework alternatives like Next.js[4].
- **Bad Decisions:** Including weak and complex abstractions code that does not make sense anymore. The team might know a better method, however, the code was not refactored to express the new optimal way of doing things. Bad decisions can be expressed in a variety of ways like poor data schema design, poor input validations, wrong assumptions, poor error handling, and lack of observability.
- **Technical Limitations**: Legacy systems often have limits imposed either by the technology used and/or the bad decisions made over the years. For instance, if we have a desktop system, we cannot make that system be accessed via a mobile application.

Legacy systems deliver value

Legacy systems are not the most desired piece of software for engineers however, for companies, they often add value. A good starting point would be acknowledging that legacy systems have good traits; there are many problems related to legacy systems, but there are good things in their favor, such as:

- **Profit**: Legacy systems are not hypothetical; they live in production. You might not like the code, technology, and decisions, but you must admit that legacy systems are how your company makes money. Why is this trait essential? Whatever you do, you must drive profits, with real business value and perceived customer impact.
- **Active Users:** Legacy systems in production have users. Having real users doubles down on the system value, presenting important future opportunities in the sense of growth. Let's say your company has a legacy desktop system with 1M users, now you build a mobile solution, and you have 150k users. The legacy system presents a user base that can grow to 850k users in the new mobile system. It's only possible to have users when we solve a real problem.
- **Baseline Testing:** It's non-obvious, but there is a testing benefit. Legacy systems are notorious for having poor coverage or no coverage at all, so you might be

3. Wikipedia definition of legacy system https://en.wikipedia.org/wiki/Legacy_system
4. Next.js by Vercel - Javascript and Typescript web framework https://nextjs.org/

wondering how testing can benefit them. When you write or have a new system, you can compare both systems. You can check how much the new system deviates from the old one. Yes, you still need to write such tests against the old system, but you have a baseline for measurements.

- **It works(-ish)**: A legacy system wouldn't have survived or had users unless it did something right. A happy path that delivers value and works consistently. Of course, surrounded by landmines, but you can learn something from a hardened process.

Not all legacy systems drive profits, but when that happens, such systems are often called core business systems. Not all legacy systems have a huge user base and are actively used; in that case, it's an opportunity to get rid of them since their profit and usage are low. Legacy systems deserve some respect; after all, they drive profits by solving real problems and running in production for many years.

Working with legacy systems

Considering the 90s as a starting point, it was possible to have companies that were not using software at all. Many of us did not live in such a time as professionals but had professors and friends talking about when companies were just using paper and process. Today, all companies have software; unless you have a startup, you always start with one or many legacy systems. Not only do companies have a software, but they also have scale; software does not shrink; it just gets bigger. You might be wondering where we are going with all this. There is no escape from legacy systems. No matter the company you go to, you will find a legacy system. Therefore, there is also no escape from monoliths and distributed monoliths.

Legacy systems are the norm, the reality, the default. We need a better approach. Otherwise, our lives will only get worse. The million-dollar question is how you will approach legacy systems properly. Continuous Modernization is the way. However, let's not jump to the solution yet. Let's understand the problem a bit more.

Amplifiers

Users want to solve problems and achieve their goals, no matter whether the system is legacy or not. Companies still need to deliver customer value and good experiences to users and make a profit. By definition, if the company wants to survive, it needs to keep involving legacy systems, creating new products that depend on legacy systems either by creating new features, fixing bugs, or keeping up with compliance. Therefore, there is a constant need for change in legacy systems. Some factors and social phenomena amplify problems that legacy systems have, forces and factors like:

- **Engineering turnover and group fear**: Engineers do not like working with legacy systems, and they quit, creating a turnover problem. Then, you have fewer