October 2022

The Legacy Trap

How to modernize applications that are holding you back, and why you need to start right now

By Michael Coté and Marc Zottner

vmware®

Table of Contents

Introduction: Caught in the legacy trap 3
Story time: Creating new opportunities in insurance 5
What is modernization? 6
What modernization means 8
Types of modernization 8
Focus on business outcomes 11
Outcome-led app portfolio rationalization 13
Find the applications 18
Bucketize and organize information 19
Jump-start monolithic modernization 30
Common missteps with traditional approaches 37
Building a modernization habit 42
About the Authors 43

Introduction: Caught in the legacy trap

"Oh, that's the mobile team. They can do anything they want." This is what I heard from an enterprise architect at a large enterprise five years ago. We were discussing their company's complicated IT systems to handle all aspects of their business, from selling to customers, handling the logistics of their service, tracking the revenue, and so on. In their office, a whole wall was taken up with a diagram that looked like a cross-section of a city's underground piping and subways, complete with bubble gum and string on water pipes keeping leaks at bay.

I'd said that I used this company's app weekly and that the app always had new features and made my experience better. I liked the company because of that app, and it made my life better. My colleagues and I were there to sell our software development platform (now called VMware Tanzu® Application Service™).

The architect was friendly enough, but they turned sour once we sat down at their desk. As with so many meetings I had back then, the challenge was something like "all your agile, cloud native stuff sounds great, but I bet you can't solve this mess we're currently in." If I could read their mind, I think they'd be thinking, "Also, thanks for getting my boss's boss all excited. Now I have to deal with you."

They were right in their sentiment. Our technology was fine, and was indeed used by many mobile teams. That mess of piping was the most critical problem. All that bubble gum and string was holding it together. These companies could eke out a few more years of evolving their business by setting up those special mobile teams, doing greenfield software. But I'm now seeing those pipes burst as the bubble gum and string crack and fray.

Most organizations have hit the legacy wall. They're now unable to change how their core business works because their IT systems are too old and unchangeable. "Most of our software development is supporting legacy capabilities that are in-house and don't provide a competitive advantage," as one executive put it, "which is opposite of what I would want to have[1]."

Organizations are now caught in a legacy trap. There's no more room on the margins to avoid modernizing their core systems. When you're caught in the legacy trap, you can't add new features to your software fast enough, which means you can't change how your business operates. You can't innovate at sustainable speed.

---

1. Forrester Consulting, commissioned by VMware. "To Recover From The Pandemic, Automate Operations To Free Budget For Innovation." September 2020.

Escaping this legacy trap is one of the most important business problems right now. It's no longer an IT problem but something that requires the attention of the business.

Colin Bryar and Bill Carr tell the story of Amazon escaping the legacy trap in *Working Backwards*. Amazon is widely admired for it's software approach and its business results show why. However, sometime in the 2000s as the authors cover, Amazon was trapped in the legacy trap: there were too many dependencies that were slowing down the team's ability to get software out the door and quickly evolve the business. As the authors explain, Amazon explored many process options (resulting in several of its well known practices), added new enterprise architecture requirements for API creation and usage across the stack. This wasn't an overnight project and took "several years of intensive and delicate work[2]."

That work at Amazon also demonstrates that the legacy trap isn't just about modernizing the hardware and software stack, it's also about modernizing the "meatware" stack: how people work together and how your organization is structured. You're not only breaking apart the core system monolith, you're transforming the organizational monolith and the way it generates software.

In my (Coté's) last three books, I discussed what this new stack looks like and many tactics for going through digital transformation. Like the organizations now caught in the legacy trap, I've skipped over a discussion of modernizing old systems.

Since the legacy trap is, my co-author Marc Zottner and I believe, the most pressing issue for enterprises now, this book covers how companies pry the trap open so they can get back to business innovation. As you'll see, there's an underlying technical collection of patterns, but the critical first step is starting with a focussed strategic analysis of the current state of your organization and its IT systems. After that analysis, the IT and business leaders then need to prioritize and actively manage the modernization program. This process is fast, holistic, and a true game changer in how you approach software.

We've worked with numerous organizations on this topic, and this book draws from that work. We'll cover the common stories, practices, and mindsets organizations use to escape the legacy trap.

At the end, all of your problems won't be solved, of course. But you should have a good idea of where to start, how to put together your strategy, how to manage your modernization programming, and how people work day to day efficiently to modernize your software.

I'm hesitant to spend much time on the urgency of this problem. When your house is on fire, you know it. Sometimes though, you get used to the flames or don't see them. However, because a large part of escaping the legacy trap is realizing that you're suffering a business problem rather than just a technology problem, let's look at an example.

---

2. Bryar, Colin; Carr, Bill. *Working Backwards* (p. 69). St. Martin's Publishing Group. Kindle Edition.

Story time: Creating new opportunities in insurance

The problem with being a mature, global company is that you're often so successful that there are few new customers or problems to solve for your customers: you've already saturated the market. We see this at insurance and telco companies a lot. Although there may be numerous houses and cars to insure, most of them are already insured. In such mature businesses, the way to grow share price is finding new revenue and driving down costs—the basics.

Let's look at a theoretical example of that business problem in insurance. Let's say the Mideastern Warm Smiles Insurance Company wants to grow revenue. They've done a great job over the past 140 years insuring houses and cars, and grew revenue in the early 2000s by acquiring a point-of-sale warranty business.

Business has slumped now, and the share price is boring. A management consulting company compiled a large report that suggested three pillars for improving shareholder value. One of them was to enter new types of insurance. Playing off the synergies of that warranty business, the consultants suggested entering short-term insurance: coverage that would last for 24 hours or less. For example, a customer might go to the beach for the day and want to insure their new, $1,300 iPhone against damage and theft. The likelihood that anything will happen (especially if they have newer models that are water and sand resistant) is low, so collecting the $50 for that one day of insurance is almost like free money to the insurance company. Now, imagine if that happens thousands, hundreds of thousands of times a day, globally.

After some business and actuary work, the CIO is given a new set of applications to create. First, the application for the insurance; next, the actuary back end for approving insurance; then, the account management for these policies. Seems simple enough: it's software. You can just add a feature.

A situation like this is where the legacy trap is often sprung. Developing the actual web app for an application is often easy, but integrating with the existing back-end services is often near impossible. In the case of Warm Smiles, the back-end systems that maintain policies only support annual policy terms. That will need to be updated. Payments must be done through bank ACH or physical checks. (Here, the CIO thinks, "Well, it did always seem weird that customers had to fill out a PDF to pay their policies.") These payments can take up to 10 business days to clear. Warm Smiles will need to modify their payment processing system. And what if someone actually loses their iPhone and wants to file a claim? Usually, claims processing takes five business days and requires an adjuster visit, but we're just talking about an iPhone here. Our CIO friend also keeps hearing the mobile team say something about batch jobs and something called an enterprise service bus. Those need to be updated as well. And so on, and so on. Meanwhile, Warm Smiles' rival, Southeastern Friendly Pats on the Back Assurance Group, has launched their own short-term insurance business and has seen a 3 percent rise in share price.

This is a made-up example, but this kind of situation repeats itself over and over in large organizations. Warm Smiles is squarely in the legacy trap. Their software portfolio does not support the business fitness and agility needed. At some point, the portfolio was great—the company has saturated their market and survived for 140 years. But, slowly and then all of the sudden, their portfolio became legacy and urgently needed to be modernized.

Sadly, most organizations have to suffer a Warm Smiles scenario before they discover the legacy trap and then do something about it. This book is all about getting unstuck from that first trap and then changing your thinking and process to prevent stepping into another legacy trap.

What is modernization?

How software goes bad

To understand what modernization means with respect to software, you first need to understand how software goes bad. Describing the way software deteriorates is tricky. Unlike your body, software does not age. It doesn't become brittle, break, get forgetful, or develop poor habits like eating salted butter on a spoon. Software can get "sick" if a virus infects it. However, a software virus is changing or faking out the original software—left on its own, software will keep running. Unlike the hunk of meat known as the human body, software does not deteriorate.

Instead, there are at least three[3] ways software goes bad:

1. **Fitness:** The software is no longer able to scale reliably to keep up with user demand. It runs too slow as measured in actual response time (time to click to new screen) or in workflow processing time (24-hour batch jobs instead of seconds). This category might also include the inability to apply security fixes or upgrades.
2. **Capability:** The software cannot support new capabilities, such as programming languages, API access, user interface types (like mobile). It can't run on new types of infrastructure.
3. **Forgotten:** People have forgotten how it works. The software could be fully capable and you could even make simple changes to update it, but there are few people with skills to operate and change the software and/or knowledge of the original designs. This means you may not know how to run or modify the software.

Reducing, or controlling, costs is another reason people often want to modernize software. However, if you can easily change the software to a cheaper version, you likely won't think of the software as legacy. If the software has one of the above ways of going bad and that slows you from changing the software to a cheaper alternative, then you'll be in a legacy trap.

---

3. There are many other reasons, especially if you ask programmers. See "Defining Legacy Code" by Eli Lopian for one list.

Here are two shorter definitions:

1. "Legacy technology is any technology that makes it difficult for organizations to change their application systems to support changing business requirements. And, therefore, it impedes business agility," said Anne Thomas, distinguished research VP at Gartner, in an email to CIO Dive[4].

2. "Legacy code is code without unit tests," according to Michael Feathers[5].

These two definitions describe the outcome and the cause of legacy software: The business can't change and there is no way to test that changes still work. The pithy Feathers definition gets to the fear part of legacy software: Sure, we could change it, but we have no way to know if it will work. His definition is also handy because it allows you to predict which software is legacy (lack of tests) and it implicitly tells you how to fix it (write tests).

All of these definitions have something in common: Legacy software is software that you have to change but are afraid to change. Otherwise, you would just call it your software. For most organizations, this means your business can no longer be changed, and you can't keep pace with the customer demands and competition.

This is where the "trap" of the legacy trap comes in: You rarely are aware of legacy software until you urgently need to change. And the time it takes to modernize that software is often longer than you have. If only you'd have known earlier. But before you needed to change how the business operates, the software was perfectly fine. As we'll see, eventually, the long-term fix for legacy software is prioritizing the ongoing modernization of your applications. That is, maintenance is a high priority, even business critical. In other words, you are stuck in the legacy trap, when the pace of change needed by the business overweights by far the application speed to market.

While you, the executive, may not have known you were in the legacy trap, the frustrating thing is that many other people were well aware of the coming need to modernize your software. There are often enterprise architects and engineers who tried to warn management for a while but have grown tired of playing Cassandra. And, of course, how often are people rewarded for pointing out a problem in past decision making? More than likely, you, the executive, need to force this issue, to ask and welcome the bad news. Perhaps it's time to talk with some senior engineers and architects and ask them about their legacy worries. James Copeman points out patterns to look for when doing this inquiry: How often has your staff changed the core systems that your apps rely on? Are they instead adding layers on top of that system, and layers on top of those layers? If your staff hasn't updated the core systems in a while, you're likely close to, or deep into, the legacy trap. This is where you need to step in and set priorities to fix these legacy problems.

---

4. CIO Dive. "Overcoming legacy debt is a process problem, not a modernization one." Katie Malone. August 24, 2021.

5. Michael Feathers. *Working Effectively with Legacy Code*. 2004.

What modernization means

So, we have an idea of what legacy software is. Now we can define modernization. While the term "modern" seems vague, it is likely that every person you chat with will have a slightly different definition. In the minds of the application executives we work with, we see four types of desired transformations emerging:

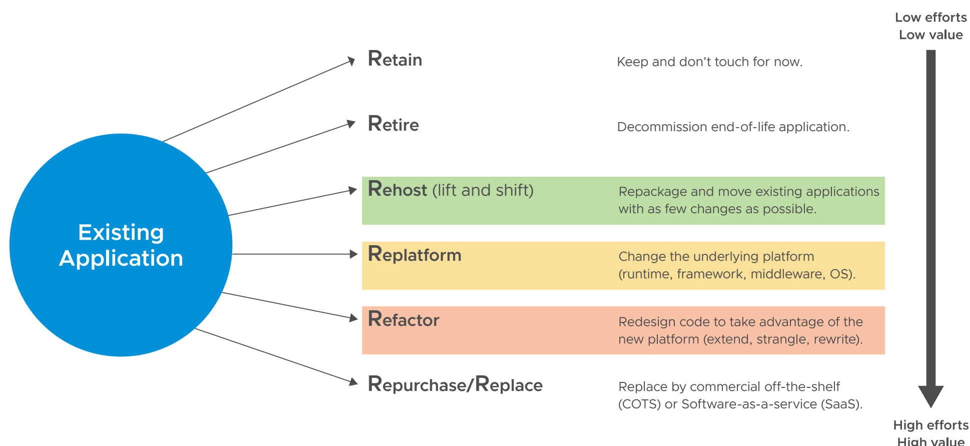
- Making apps better cloud or dematerilized data center citizens
- Evolving the application architecture (Data/Event/Test/Domain Driven Design, microservice, multi-channel, responsive)
- Improving collaboration, security, and governance along the app lifecycle (lean product management, agile practices, Dev/Sec/Biz/Network/Fin/Ops)
- Making apps more intelligent, deterministic, configurable (automation, machine intelligence, rules engine)

Primarily, modernizing means making whatever changes are needed to make adding new functionality to the software easy, quick and affordable. This means changing the design of the software, writing new code and tests, moving the software to new infrastructure, or rewriting the software from scratch. Secondarily, modernization can mean optimizing to lessen the negative impact on time, money and your attention. Maybe you can't get rid of the legacy application, but you can make living with it less terrible.

Let's look at the major types of software modernization.

Types of modernization

So far, there are seven ways of modernizing any given app. Perhaps humanity will one day discover an eighth, ninth or even tenth if we survive long enough, but these seven come up over and over. By design, these seven types of modernization all begin with the letter R so that we can label them the seven R's. Here they are, ordered from least to most effort, risk, and value[6].



6. Several of these R's have been used in the industry for many years. We have built these R's on those and also our own work. Chris Swan documented many of the R lists and variations, from the 4 R's to the 7 R's, to some other R's. Most famously, Richard Watson published a 5 R's paper at Gartner some 10 years ago and has been updating it since.

Retain

Keep and don't touch the app for now. That is, make no changes and keep things running as they are. This is probably the default option for most apps in your portfolio. Doing nothing can be a wise strategic choice.

Retire

Decommission an end-of-life application. That is, shut it down. In this case, your analysis often finds the application is used very little, has been superseded by another application, or is no longer profitable to run. A good example here is the Minitel service, once the world's most successful online service. Once the internet gobbled up all of "online," Minitel was finally retired after 32 years of operating in June 2021. Applications that were purpose-built for regulations that no longer exist are another common app to retire, as are applications that run parts of your business that no longer exist.

Rehost

Often called lift and shift, this is repackaging and moving existing applications with as few changes as possible. This is sort of like just copying an application and all its data to a new computer. Typical examples are cloud and data center migrations or the process your company has been through while virtualizing its data center.

Replatform

Here, the application remains the same, but there are significant changes to the underlying technology stack and platform (runtime, framework, middleware, operating system) of your app. This can require changes to the actual application itself, but they should be very minimal. For example, replatforming might mean moving from an Oracle WebLogic Server to Spring Boot, from .NET to .NET core, from Windows or AIX to Linux, or moving your application from virtualization to containers.

Refactor

In this type of modernization, you're finally changing your application's code deliberately. When you refactor, you redesign and rewrite parts of your application to take advantage of new platforms and capabilities. This is distinct from rewriting in that the functionality of the application stays the same and isn't updated; just the internals of the app are changed. This is sort of like keeping the exterior and interior of your car looking and operating all the same—hopefully a powder blue 1980 Monte Carlo with a dark blue interior—but replacing everything under the hood and under the body. For example, you might refactor your application to scale from thousands to millions of users as your business gains customers. From video game back ends (e.g., Diablo II) to core banking systems (e.g., the open banking evolution) and governmental services exposed to citizens over the internet (e.g., the German Online Access Act), this option is often the default cost-effective choice to rejuvenate existing systems while bringing them to a new era.

Rewrite

The name says it all: You restart from scratch and write a new application the way a thriving software company would. Your organization still needs what the application does (for example, registering for fishing licenses or scheduling ice machine maintenance), but the old application no longer solves the problem well and is challenging to maintain or evolve. Instead of just duplicating the same screens and workflows but with new fonts and colors, this type of modernization gives teams the chance to reimagine how the application functions. Rewriting carries the most risk but will bring the maximum outcome. You're given the chance to newly understand the problems your company addresses and the way your software solves them.

Replace

In this scenario, you still need the functionality the application provides, but you no longer find value in owning the application. Instead, you outsource it by replacing it with a commercial off-the-shelf (COTS) application, often a software-as-as-service (SaaS) solution. The same outcomes are achieved, but you now use a third-party or outsourced option. Your development teams can now focus on the apps securing your competitive advantage.

Such transformations are straightforward for highly standardized systems, such as mail or file servers. Also, while remaining in the same software vendor's ecosystem (e.g., Office to Office 365), replacement paths are frequently covered by exhaustive guides and tools.

For non-standard, end-of-life systems, this is often the most effort-intensive option. For example, transitioning your highly customized enterprise resource planning (ERP), customer relationship management (CRM), human resource management (HRM), or e-commerce system to another will likely be a daunting task. The effort is usually worth it, however, as all that customization, stockpiling over the years, becomes a boat anchor that's causing all your problems.

Prioritizing with technical and business drivers

Once you start identifying software to modernize, you'll soon discover there's too much of it. Organizations often find there are thousands of applications and services on the list. Air France-KLM, for example, has been working on modernizing 2,000 applications. Larger organizations could have even more, especially after many years of mergers and acquisitions.

Putting together a plan for what to modernize is confounding and endless if you only focus on technical drivers to modernize. As you assess whether to modernize each application, the answer will frequently be "yes" because, for the most part, there's always a better way to run and build applications. This quickly creates a large, overwhelming list.

Modernization project failure is often caused because there is little to no focus and alignment on the business drivers and goals of modernizing the applications. We find that most modernization strategies fail because they focus too much on the technical aspects and not enough on the business-related aspects.

So, in the next section, let's look at the critical role that linking business value to modernization plays. Now that we know the "what" of modernization, let's look at the "why."

Focus on business outcomes (or, business in the front, modernization in the back)

One way to think about the legacy trap is to finger-wag at managers who deprioritized the needed maintenance work. Instead of spending time and money on upgrading old services or rewriting applications in newer architectures and frameworks, they chose to bolt on new features or just chose to do nothing to save money. This is not always a fair assessment.

You might also be in the legacy trap because the decisions that slowly walked you into that trap made perfect sense at the time. Here, we think of two types of applications that most people use frequently: filing expenses and filing health insurance claims. In both cases, the software often feels ancient, dated, and tedious. It doesn't help that the workflows in each are essentially justifying why you should get money with the very real prospect of people telling you no for weeks on end until you unlock the riddle of how to categorize your requests and include the proper paperwork.

While the experience may be bad, those businesses likely had no urgency, no need to modernize their user experience (UXs) and interfaces (UI). In both cases, most people don't get to choose the software they use; the company they work for does. So, there's no need to compete on your feelings and productivity. Indeed, the actual customers of these applications are often people who don't actually use the software: large organization managers who, at worse, have admin assistants to handle such annoyances. Unless modernizing your software directly addresses a business need, you won't get far. And maybe that's as it should be.

The error in incorrectly prioritizing application modernization in favor of other activities, such as adding new features, often has to do with deferring suffering to the future. You trade short-term benefits for an intangible debt in the future. Indeed, the term tech debt is used to describe the mounting deferral of maintenance tasks. Just as taking on too much debt can bring down a company, taking on too much technical debt will grind your business to a halt. Recall the short-term insurance example from the start of the book: Until the business need to grow revenue came up, there was really no reason to modernize the software stack that supported the current, successful business.

Navigating this line between taking on helpful debt and too much debt is a rare skill. Analogous to actual debt, a good executive, supported by their team, will know when to service tech debt by slowing down feature delivery to ensure long-term flexibility.

This gets us to our first principle of escaping the legacy trap: The only reason to modernize your software is to give your business new capabilities to deliver on new business ideas. Instead of proposing that you need to modernize your software estate simply because it's old, start from the why: Your business (now) needs flexibility and speed to change and innovate, so you're going to need similar capabilities from your custom-written software. This will involve updating some of the services, adding new APIs, and maybe even replatforming to new infrastructure. The trick is to position modernization as simply part of the normal development process.

Doing this for a large, highly visible project—such as the core workflow for renting cars, for example—is high risk and stressful. Instead of waiting for the sudden need to modernize everything all at once under time pressure, you should focus on regularly, proactively gardening your application portfolio. Pick apps to modernize that are closely connected to the business, following the same approach of modernizing for the sake of new business capabilities but with lower risk. You pick small, meaningful apps to modernize instead of big ones. We'll talk about the method we've used with hundreds of organizations in a little bit. First, let's talk about why you take this approach.

Why nibbling at the edges is better than taking big bites

Modernizing software is a difficult task. As with so much programming, the actual designing of the app and writing code is the least of the problems. The difficulty comes in managing the integrations and dependencies on other services, and the way in which the software is configured and run in production. This is where legacy software is particularly thorny. Most organizations have poor visibility into how changes in one part of their portfolio will affect other parts of their portfolio. You can't just change customer or inventory databases, for example, because so many other apps depend on the way that data is structured and accessed. So, you have to be careful and deliberate. Though it may sound professionally crazy to say it out loud, you have to be prepared to fail and roll back changes. In short, you'll need to learn a lot, find out where the pitfalls are, and adapt your plans accordingly.

This is why starting with smaller, lower visibility apps is key. These apps should be customer (or internal employee) facing but should have as few dependencies on other apps and services as possible. When your changes to these apps cause problems or schedule delays, the effect on the business will be minimal. This gives you the chance to learn firsthand what modernizing in your organization will be like. Who do you need to talk with to get networking access to run a new access layer on your ERP database? How long does it take to get a meeting with them, and do you need to cajole them to help you? How will you deploy the new app into production, and are there governance processes you need to go through to launch in the EU? These are all things outside of the actual programming and testing that you'll need to figure out as well.

In short, you have to give yourself time, space, and safety to learn by failing. That's why you proactively pick small but meaningful modernization projects to get your teams trained up for the big apps when they come along. This is a major mindset change when you're escaping the legacy trap: You're not just changing applications, you're establishing new organizational capabilities.

Common business outcomes used to drive modernization

Justifying modernization because the business needs to launch a new app or new features to support a new business or modify an existing one is straightforward. You can attach the modernization project to the business strategy and case. However, there are other, very common business capabilities that justify modernization.

Amy Hertzog and Corinne Dubois cataloged these common drivers:

Creating an innovative, customer-first corporate culture

- Technology investments might center on automating a pipeline to production, creating a new asset inventory system to streamline the decommission of unused resources, or something entirely different.
- Investments in employees might focus on upskilling project managers to product managers over specific agile framework training, or vice versa.
- Efforts to change the organization's culture might focus on creating a culture of quick feedback cycles or a cross-functional partnership (or both). It might focus on heavy use of outside consultants or a slower, internally grown set of coaches.

Better customer responsiveness and time to market

- Technical solutions should minimize barriers to releasing code, such as continuous integration and delivery (CI/CD).
- Efforts should ensure developers have incentives (and training) to connect with the customer and deliver software in smaller sections of value.
- Culture change should focus on moving development teams from fixed projects measured on cost/time constraints to customer-focused products measured on objective or outcomes achievement.

Cost reduction

- Implement technical solutions that help automate some of the control guarantees the organization needs.
- Improve relationships and processes for the people who need to coordinate across organizational boundaries.
- Institute a culture of transparency around prioritization based on cost-value analysis.

You can use this list of common drivers to identify reasons for modernization that are not as straightforward as creating a new app or features.

Outcome-led app portfolio rationalization

We now have the guiding principle for application modernization: What you select to modernize and when is driven by what your business needs. Now it's time to go over the actual work of modernizing your portfolio. This chapter will walk you through the key activities and vital concepts to build and execute an efficient transformation strategy.

First, we'll look at the three core components of a portfolio rationalization strategy. Second, we'll expand on how to quantify the desired outcomes and build a business case for the modernization strategy. Finally, we'll walk through the process for selecting and then modernizing each application: finding and grouping all apps, elaborating a decision framework, iterating on the code, and continually using feedback and real work to adjust your strategy.

Strategizing portfolio rationalization

Strategy means many different things in different contexts: how a business will achieve competitive differentiation and revenue, principles to follow and constraints to work in, quantifiable outcomes we want, or just a simple, ordered list of things to do (a plan). For us, a modernization strategy answers the three following questions:

1. **Disposition:** To what degree should I transform my apps? For example, should I completely rewrite or just move them from on-premises to the public cloud?
2. **Stacks:** What are my target technology stacks?
3. **Prioritization:** Which apps and teams should I transform first?

Answering these three questions is the start of creating your modernization strategy. Let's look at each.

Disposition

Identifying the right level of modernization, or disposition, for your apps is crucial to maximizing the value you get out of it. It makes no sense to transform every single app into its better self. Although the target disposition might seem obvious for technology-driven, horizontal modernization programs, it's never the case.

As developed previously, we have seven disposition options for each application: retire, retain, rehost, replatform, refactor, rewrite, or replace your applications.

Stacks

Thinking through where and on what your apps run is a key consideration. By stacks, we mean technological things, such as public or private cloud, or on-premises; mainframes instead of virtualized x86 environments; different programming languages, frameworks and runtime stacks; and architecture models, such as microservices, instead of monoliths.

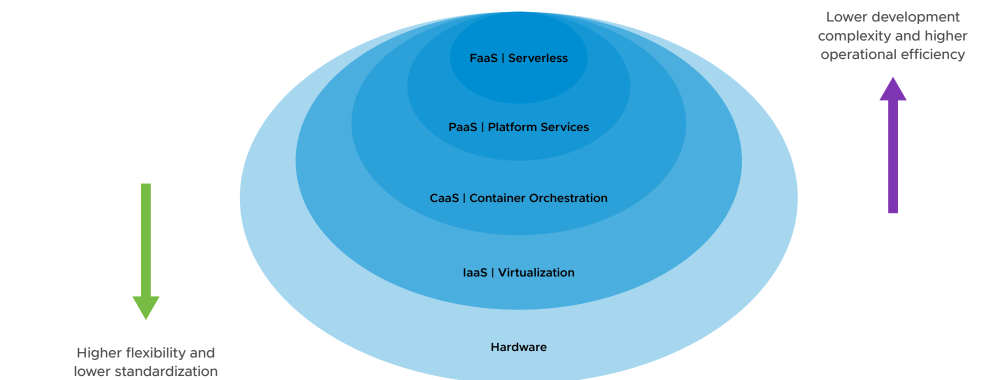
Focusing on the infrastructure layer below the application may seem too deep down the stack, sort of like worrying about what type of lumber you're using in your house's rebuild. However, just like the type of lumber and piping you choose, choosing the platforms your apps run on is incredibly important. Platform choice will drive costs and will also determine (or limit) your application's capabilities and how the application is managed.

Such stacks typically comprise the following:

- A distinct infrastructure-as-a-service (IaaS) layer (e.g., landing zone on hyperscaler, on-premises, hybrid models)
- An application platform, such as platform as a service (PaaS), container as a service (CaaS), or function as a service (FaaS)
- A software development stack (programming languages, frameworks in use, components, and even development tools), including design principles (event driven, microservices, cloud native)

As shown in the following diagram, it is all about choosing the right abstraction layers for your development and operation teams. Strategically you would want to push as many workloads as technically feasible to the top of the abstraction hierarchy.

Stacks: Available Application Abstractions



Prioritization

Aiming for the highest impact with the least effort at the beginning is a great idea. Such prioritization can only be achieved by deeply linking transformational work to your desired business outcomes and objectives. With this approach, you increase the chance that you transform the right apps before you transform your apps right.

Starting with the desired business outcomes: Quantifying the business value of modernization

Many of the IT departments we supported had a hard time convincing the business to let them modernize existing applications. Efficient and impactful modernization efforts require alignment with well-defined business outcomes. Otherwise, you'll likely end up addressing painful IT symptoms instead of business problems: replacing one tech with another, adopting a new platform, or transforming architectures but then struggling to deliver business value. You won't be able to make a business case or will solve the wrong problems well.

Every modernization initiative should start by understanding your organization's current business challenges and defining the desired outcomes. Here as well, we want to begin having the end in mind, to paraphrase one of Stephen Covey's habits of highly effective people.

The set phrase "business outcome" is a generic concept applied to all sorts of organizations. In general, all businesses want to increase revenue, retain existing customers (and get more money from them), and increase profits often by cutting costs. Telling you that, though, is like saying water is strongly recommended as a major part of your diet. Many people find it difficult to directly connect changes to their software to business outcomes. This is especially true if software has been delivered as a series of projects rather than an ongoing product[7].

When you can't connect your software to business outcomes directly, you'll need to focus on the capabilities your software provides to the business. This will allow you to evaluate your portfolio based on the business outcome the software helps make possible. When it comes to transformation initiatives, these capabilities are usually speed, savings, security, stability, scalability, and shining in this process. Let's look at a summary of each of these and the capabilities they provide.

Speed gives you the ability to release software more frequently to learn what features are the most valuable. It also allows you to keep the software up and running because you can respond quickly to problems:

- For developers, speed means less time between the idea to running in production. This means shortening the release cycle. The business outcome is faster time to market.
- For operations, speed means less time spent planning, approving, and building infrastructure. It also means faster times to diagnose problems and, critical to DevOps and site reliability engineering (SRE), faster time to recover from errors. This usually means focusing on a huge degree of automation.
- For product managers, faster cycles give them more feedback on how well the software addresses real customer problems and solves them. The frequency of feedback is often unappreciated: More feedback and faster learning means the organization becomes smarter[8].

Saving money by avoiding and reducing costs is often the top topic for major modernization projects. Savings are achieved by conducting one or several of the following:

- Replace proprietary, costly software with generic equivalent technologies to reduce the Total Cost of Ownership (TCO)
- Optimize how people and organizations work. Most organizations follow zombie processes that are no longer required or could be automated and sped up. or example, on one project, the US Air Force was able to speed up releases from years to three to six months after automating many steps of compliance policy checking[9]. This type of savings is often achieved by changing how people work together and adopting standards across the organization.
- Staff are more productive, experience less wait time, and otherwise can develop code to run the business.

7. For a longer discussion on the difference between project and product mindsets, see Changing Mindsets.

8. For a longer discussion on this topic, see Monolithic Transformation.

9. For more, see around 18:00 into the talk "AUSA 2021 Warriors Corner: By Soldiers, For Soldiers: Building an Organic, Soldier-Led Software Development Capability in the Army," October 10th, 2021.

Security is table stakes for any application. Of course it needs to be secure. However, you can track the strength of security and your ability to be responsive and resilient in security. In short, there are often ways to do security better:

- Follow the three R's of security: rotate credentials to prevent malicious access, repave your applications to return to a secure state, and repair them by quickly deploying patches.
- Use DevSecOps tooling and thinking to have more confidence in your risk assessments and verification of software builds.
- Increase collaboration between developers and security staff to work closer together.

Stability means your application works. It's available and keeps running with little or minimal interruption of service. Once modernized to a cloud native architectural style, your applications will have several new stability capabilities:

- The ability to deploy new versions of your software with rolling updates that slowly move users from the previous version to the new version, preventing downtime and maintenance windows where the software is unavailable
- Better high availability and failover between different data centers and cloud because of the cloud native stateless architecture and emphasis on multi-cloud standardization
- Fewer configuration errors between development and production because the environments are either nearly similar or exactly the same

Scalability is the capability to handle growing numbers of customers and usage. It's the business capability to grow. Stability and scalability are closely related, with each enabling the other. Better scalability offers several benefits:

- Savings through better resource utilization: If you can scale up quickly, you can often scale down quickly—which means paying less if you're working with cloud pricing.
- Better capacity planning: This leads to cost savings but also to the capability to handle growth.
- The ability to add more computing power across different data centers and clouds: This improves stability as well as speed.

To use the six S's to link your software to business outcomes, you'll likely need to quantify them. There are some obvious ways to do this; for example, the speed of software releases, how quickly and often you can deploy security fixes, and money saved. However, you may need to come up with some custom metrics as well[10].

Start small and iterate

You're probably thinking, "This is a lot of work when multiplied over thousands of applications." But you can force yourself to make it simple and quick to show business value if you work iteratively instead of modernizing all of your applications at once.

10. For examples, please see the S's framework and Coté's talk "Beyond DevOps Metrics."

We've used a simple, short workshop format with hundreds of organizations to start modernization programs and start delivering modernized apps in 10 weeks or fewer.

These inception workshops bring together the transformation leadership team, putting business, architecture, program, operations, development and management all in one room to work as efficiently as possible. In this time frame, the team explores goals and anti-goals. This helps bootstrap the overall modernization strategy by creating concrete objectives with associated key results (OKRs). These goals will be the north star of your transformation team to ensure that all time invested contributes to the greater business good.

Once your goals are defined, the next step is to figure out what application to start with. This is a key part of how we've seen organizations successfully modernize their applications: They start with one application, not all of the applications at once. This can seem risky for a business. You'll be under pressure to make big promises during your annual planning cycle and show that your business case has a large payoff, a strong ROI.

You and your management are probably thinking that if every developer team does two months of training and planning, you can get all your groups working in parallel and modernize by the end of the year. Promising that you'll modernize hundreds, worse, thousands of apps in the first year is a terrible strategy and will likely fail. Instead, you need to start small, deliver one or two apps, learn and adapt, grow a bit more, iteratively learn and adapt, and so on. This will take time, likely months. But instead of overpromising and failing, you're more likely to succeed[11].

Let's next look at how you find the initial and subsequent applications to modernize.

Find the applications

Application is one of those words that means a lot and not much simultaneously: It's broad, not deep. Depending on what they work on, each person you talk with is likely to have a different definition. Someone who works on mobile apps will think of an application as the screens and applications a user interacts with. Someone that works on software that checks a customer's available cell phone minutes will think of that as an application even though a human is not directly involved in using it. Someone who's in charge of processing payroll changes will think of that as an application. Even worse, each of those people might use an additional name for the software they work on—"service" is a popular term, and mainframe-minded people might say "workload" or "batch job."

Applications run the business

So, let's establish our definition of application. First, an application is software that runs the business. That may be obvious, but people often forget that it's the primary guiding principle for applications. When you look across your application portfolio, you'll find many applications that don't seem to strongly connect to the business or provide real value. Applying the six S's to strongly link applications to business outcomes will help you find this real value. Focus on and remember this first principle: Applications run the business.

11. We realize that this pleading is usually unrealistic in most large organizations. Get as close to the spirit of what we're saying as possible. There are tactics for working with management and corporate finance that are obstinate to reality. For some of them, see The Business Bottleneck and Changing Mindsets.

Applications usually bring together four elements:

- User experience (UX): The provided UX makes its usage valuable, efficient, and delightful. This manifests itself in screens as well as in the workflows people go through to do something with the software. As a simple example, when filing expenses, taking a picture of a receipt with an app instead of manually entering the details is generally a better user experience.
- Interfaces: Different interfaces expose part of the application's functionality. This is often in the form of user interfaces, the actual screens, and the workflows people interact with. But these can also be services that other applications use instead of humans. In this case, these interfaces are often called services or APIs.
- Business logic: Often referred to as intelligence, business logic defines the application's core behavior and executes it. This is the code that actually does something.
- Data: This is a collection of data points and processes that the application uses, creates, and changes.

Automate application catalogs

Believe it or not, just finding all applications deployed within an enterprise is often a very daunting task. Although the concepts in application portfolio management were developed in the late 1990s, best practices are not widely followed. Despite efforts to automate asset databases, such as configuration management databases (CMDB), application lists are often managed manually, leaving the official lists incomplete and out of date. To get a full view of the applications in your portfolio, you'll often have to look over several systems, CMDB, spreadsheets, and maybe even whiteboards. In short: It ain't going to happen.

Instead of relying on manually updating spreadsheets to identify and list all existing application and infrastructure assets in a central inventory, we recommend automating the process. Tools to automate the discovery process can combine several approaches: be agentless, conduct port and packet scans, rely on automated build and deployment pipelines (infrastructure as a code, CI/CD, application repository), or run on SaaS or on-premises environments. They could range from self-written scripts, to commercial offerings.

Bucketize and organize information

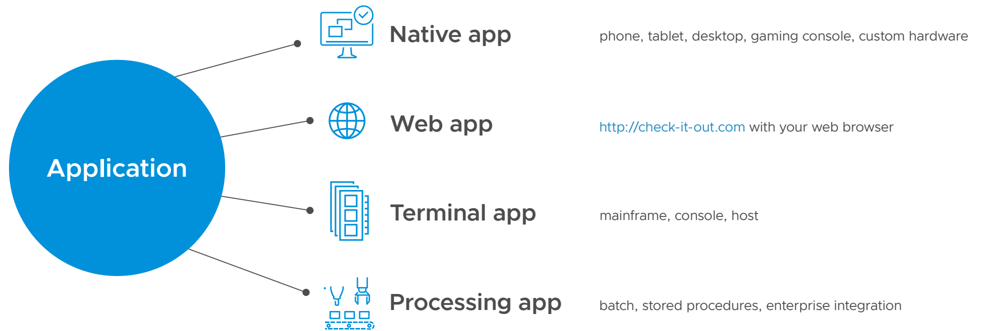
Once you've found the applications, you'll categorize (or bucketize) them based on the type of application they are and the technical characteristics of each application. Buckets group similar applications based on their business and technical similarities. With well-defined buckets, portfolios of thousands of applications are often grouped into tens of application archetypes. This allows you to analyze and work with those applications in aggregate instead of one by one. This step reduces complexity and provides a better grasp of your entire portfolio.

There are several ways to bucket apps, and you'll need to find the ones that work best for your organization. Let's look at the most common methods.

Application type

One of the most common ways to categorize applications is by type. Type is often defined by how the application is accessed and used by people. These are the most common types:

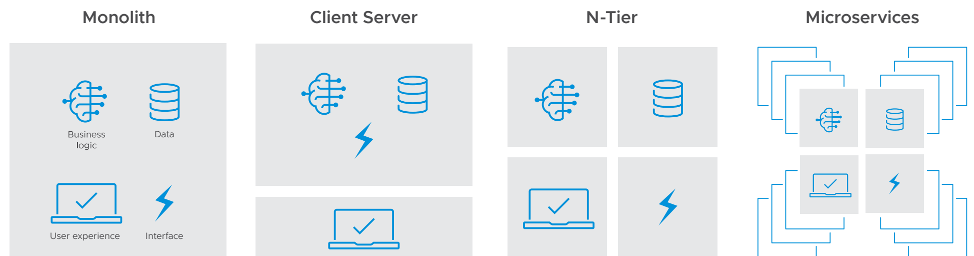
- **Native apps** run directly on a phone, tablet, desktop, gaming console, or custom hardware.
- **Web apps** are accessed by a web browser with a server on the back end.
- **Terminal apps** run on a mainframe, as Windows virtual desktop apps, or in a console without an elaborated user interface.
- **Processing apps** run as a batch process, stored procedure, or enterprise integration middleware.



Most of the time, you'll likely categorize applications by their type. It's essentially for prioritizing based on technical feasibility and risk to changing the application's source code. So, bucketizing by type is a good place to start.

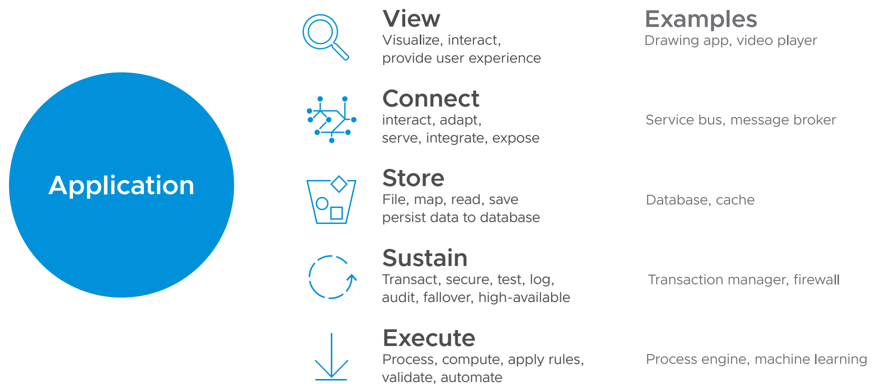
Application architecture

Another way to group applications is by their architecture. As with so many software-related terms, to developers, architecture means many things, but what we mean here is the structure and style of the application. Another phrase commonly used here is software design. Any piece of software is composed of different components, if only conceptually. Some architecture styles put all those components on one server, tightly integrated together. Others divide them up by server side and client side, and at the extreme end, a style such as microservices divides the application into hundreds—often thousands—of independent parts.



Application function

Defining what an application does, its function is straightforward when you're talking about an application with a user interface (UI). However, as mentioned above, application often means a service with no UI that's used as a component in another workflow. These UI-free applications are, without any irony intended by developers, unfortunately, called headless applications. In fact, you'll find that many of the applications you're modernizing are these headless applications. Or, they may seem like that after you use them. Here's a common list of functions:



- **View** delivers an interactive user experience and visualizes information.
- **Connect** exposes, adapts, consumes, serves, and integrates functionality through interfaces.
- **Store** reads, saves, and maps data to file, database, or cache storage.
- **Sustain** supports nonfunctional aspects of existing systems, such as transactionality, security, testing, logging, auditability, high availability, and failover.
- **Execute** processes, computes, validates, automates, and applies rules covering some business logic.

Application repository

You're almost done creating a quick profile of each application. There are other characteristics apart from the application's function and type that you'll need to gather. This information mostly has to do with the context of the application, things such as:

- Information about the application, such as name, department, and current version
- A list of key stakeholders across the application lifecycle: product owners, lead architect, test manager, business owners
- Regulations and other policy that the application must conform to
- Captured technical scores on complexity, efforts, security, and business-relevant information for prioritization purposes

There may be other attributes you want to capture, such as desired release date, usage, and so forth. What's important at this point is to construct your portfolio. Like all mission-critical work, this is usually done in a spreadsheet, as the following screenshot shows.

Custom group for similar applications

Owners could be further split between:
technical lead, business owner, test owner, external provider

Business domain	Logical application	Artifact name	Relevant version(s)	Type	Owner	In-scope ?	Target release	Complexity	Business criticality	Usage
Core	CoreBanking	bank.ear	7.3_2020.FINAL	Frontend	Jack Fruit	yes	Q4 2022	High	High	High
Core	CoreBanking	bank-backend.jar	7.3_2020.FINAL	Backend	Bob Meister	yes	Q1 2023	Medium	High	Medium
Batch	JobManager	process-engine.war	13.0.5	COTS	Adam Apple	no	-	Low	High	Low
...	...	...	...	...	...	...	...	...	...	...

Information for internal classification

Planning and modernization scope

Relevant for prioritization

With a quick-and-rough portfolio put together, you can now start to select which applications to modernize. To read more about this kind of process, check out VMware Tanzu's Rapid Portfolio Modernization process.

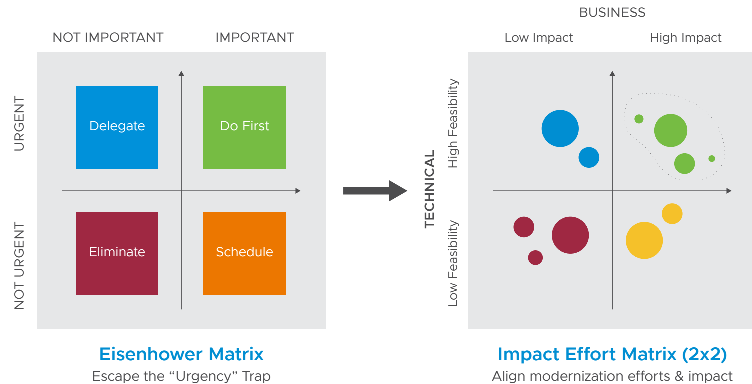
Craft a decision framework

So far, we've defined our desired business outcomes and created a list of all relevant applications grouped by buckets. Next, we'll use a 2x2 matrix, also named "impact effort matrix", to force rank the applications and determine which ones to start with first and how far we should modernize. The two factors for this are the technical feasibility of successfully changing the application and the business value you hope to gain from modernizing the application. That is, how hard will it be to change and will it be worth it?

Gauging technical feasibility perfectly is impossible until you start working on the application. However, you can get a good enough sense for your initial analysis. Typically, each application can be assessed and scored leveraging automated analysis tools that focus on several factors:

- Feasibility: What will it take to technically move the application to a new stack? For example, what will it take to move from traditional, on-premises infrastructure to public cloud? There are several automated toolkits that will do this analysis for you, such as Cloud Suitability Analyzer, Windup, and IBM Migration Toolkit.
- Security: How vulnerable are your applications through embedded third party libraries and code smells? Dependency analysis and static application security testing (SAST) tools like OWASP Dependency-Check, Find Security Bugs Snyk and Grype provide great help.
- Quality: Consider reflecting code duplication, languages used, anti-patterns, documentation, to seize maintainability and extensibility.

The business value from the transformation will help to focus on the right applications. It should be derived from the business outcomes you initially defined (OKRs on the 6S) and could reflect how differentiating one app really is for your business. For example, an app sharing the corporate cafeteria menu is less likely to have a high differentiating value for a car manufacturer than a connected-car app.

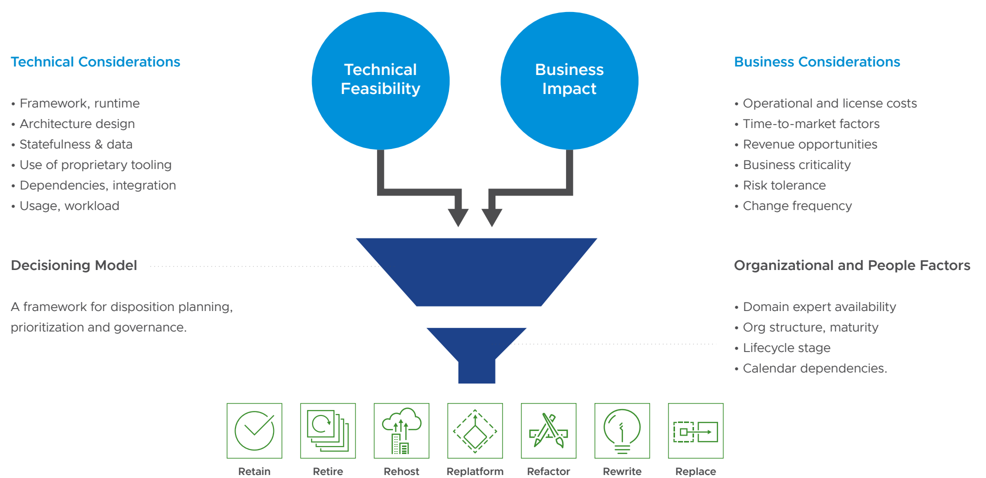


Eisenhower Matrix
Escape the "Urgency" Trap

Impact Effort Matrix (2x2)
Align modernization efforts & impact

Similarly to an Eisenhower, matrix helping to escape the urgency trap by rating urgency and importance of activities and take action, our decision framework relies on a 2x2 matrix where one axis represents business value and the other the technical feasibility to successfully change. Applications in the upper-right quadrant have a high business payoff and low modernization effort and risk. These are the best applications to start with.

We've found that in the first few rounds, mysteriously, all of the apps end up in the upper-right quadrant. This is actually no mystery: Each person tends to think their applications are the most important. If this happens, you need to zoom into just that quadrant by moving both axes to recenter the matrix. Repeat this until you end up with obvious winners instead of a stack of notes. Often, you'll find that some people will not know the priorities and import of other applications, nor of their own once they try to justify why their application is so critical. This is a bonus of the process; the team gets more familiar with the portfolio as a whole.



Such a decision framework can typically be elaborated in a few days, without spending months of analysis paralysis trying to understand the specifics of every single app in your portfolio. It's powerful to achieve alignment and transparency with all stakeholders. Far from a dogmatic and immutable source of truth on disposition, this framework should become a living, evidence-based decision guideline to drive prioritization and transformation.

The reason to start small is to learn the process for doing application modernization in your organization. There are many commonalities to that process across organizations, but organizations also have their problems and ways of solving them. So, learning from the work done should be incorporated into the next iteration. While being transparent to everyone, the modernization decision model becomes a living framework updated after each transformed application. For example, organizations we work with often discover in their initial projects that they'll need to work with networking, security, and compliance a lot more than they accounted for. They use this newfound knowledge to reassess the technical feasibility and plan the next round of apps. Organization factors, such as the availability of domain experts, team interest, project dependencies, and deadlines, are often used as a third dimension to further filter and prioritize work. Finally, this framework should be owned and maintained in-house as it defines the very essence of your modernization strategy, perfectly aligned with your business strategy.

Case study: Digital whiteboards

During the first years of COVID-19, companies were forced to shift to 100 percent remote working. For an activity like whiteboarding that depends on having everyone in one room, this would seem very troublesome. However, in being forced to find new ways of working, many organizations we work with discovered online whiteboards, such as Miro and Microsoft Whiteboard.

In our opinion, these turned out to be even better than meatspace whiteboards for modernization analysis and likely any type of whiteboard-driven analysis. This is because updating the whiteboards is easier, offers better tracking of the whiteboard's history, allows for more collaboration over time because people can add comments online at their own time, and enables much more transparency that leads to more information because anyone can access the whiteboard. Editing it, of course, is much easier than moving sticky notes around an actual whiteboard in a specific room and time zone. Also, people who are reluctant or unable to add their input can find it easier to add to a digital whiteboard.

Even if you're all in the same room, using digital whiteboard could be considered as the way to go.

Iterate and improve

After a time frame of two to six weeks of analysis work, you should have the first version of your transformation roadmap, including a decision framework defining your strategy for your applications:

- Disposition: How far should each application be transformed?
- Stacks: What are the target technology stacks?
- Prioritization: Where should we start?

With this in place, you can get the modernization ball rolling. You should assign teams to each top candidate application. As a reminder, you should only start with one application, or three at most. Learning by doing real work on a few pilot applications is, in our experience, the safest, most sure way to move forward with modernization.

Modernizing how you work (or, culture is hard)

We covered how to quickly build and validate a pragmatic, outcome-oriented rationalization strategy for your application portfolio. This is done by quickly creating an inventory, bucketizing applications, and defining a living decision framework.

A large portion of this is tied to technology and practices. But wait a minute. This is not enough if you want to change the way you build software and excel at it. Beyond code and infrastructure changes, you will have to seed a new culture within your organization to adopt a new mindset while crafting modern software. A successful transformation has to be holistic, covering not only platform, tools, and practices, but cultural aspects as well.

We'll cover this in the next chapter before digging into the most effective method to modernize legacy, monolith applications.

Escape trap: Core tenants

Many modernization initiatives focus on rolling out a plethora of tools and technologies: migrating apps from tech A to tech B, lifting and shifting workloads to the cloud, containerizing existing systems, and adding new tools to automate the DevSecOps development toolchain. Too often, the emphasis is on the solution space (e.g., using an ad hoc tool) rather than confronting the reality of the problem space.

To unleash the true potential of your application modernization strategy, you should focus on modernizing the organization structure, process, and culture. Older software keeps you in the legacy trap, but so does an outmoded organizational culture.

Changing both technology and culture is not easy. However, it's much more rewarding as it intrinsically turns your teams into highly motivated, state-of-the-art, customer-obsessed development squads. In this chapter, we'll take a closer look at concrete shortcuts you can take to progressively metamorphose your engineering culture and methods to become more of an Apple, Amazon, Netflix, Google, Tesla, or Meta.

Seed a cloud native culture

It's no contradiction to say that being cloud native does not have much to do with cloud computing. There is an idea that the cloud is a place, a suite of technologies or services that run somewhere in data centers. But the cloud is not just a place; it's also a way of working. This means that it's a new type of IT culture.

But what does culture really mean in the context of cloud and modern apps? The DevOps community uses the Westrum spectrum to categorize company cultures[12]. American sociologist Ron Westrum defined organizational culture as “the organization's pattern of response to the problems and opportunities it encounters.” This could be broken into three cultural archetypes, as the following chart shows.

Source: DevOps culture: Westrum organizational culture

12. For example, see the annual Accelerate DevOps Reports.

Pathological (power oriented)	Bureaucratic (rule oriented)	Generative (performance oriented)
Low cooperation	Modest cooperation	High cooperation
Messengers “shot”	Messengers neglected	Messengers trained
Responsibilities shirked	Narrow responsibilities	Risks are shared
Bridging discouraged	Bridging tolerated	Bridging encouraged
Failure leads to scapegoating	Failure leads to justice	Failure leads to inquiry
Novelty crushed	Novelty leads to problems	Novelty implemented

A pathological culture focuses on the personal interests and resources of the leaders. Information is processed so that it only advances parts of the organization. It’s power, political and blame oriented. A bureaucratic culture, on the other hand, centers on channels and procedures. Innocent at first glance, it becomes problematic when dealing with urgent problems or crises, or meeting needs beyond the limits of the rules and process in place. Finally, there is the generative culture. According to Westrum, a generative culture is the most effective in maintaining a highly functional organization. When it comes to cloud native, we want a generative culture.

Constructing culture

In software, a generative culture is focused on learning, ongoing improvement, and moving most decision making as close to the developers as possible. Rather than relying on centralized, long-term decision making, a generative culture pushes decision making and responsibility to the team working on the software. The people on this team should be the best informed about what people do with their apps and how the apps could be improved.

Shifting responsibilities down the management chain means changing the response to failures. Failures and accidents are seen as opportunities to improve, not witch hunts. Errors are subject to honest postmortem, and risks are shared. For example, Netflix calls unwanted incidents surprises, eliminating negative stigma and encouraging people to bring mistakes to light so they can be learned from. Strong cooperation is encouraged, with cross-functional collaboration toward common goals. There are no heroes but rather a culture that embraces individual strengths to create cohesive teams.

In fact, many cloud native companies—those that build and run scalable applications in the cloud—make it a point of honor to publicly define the principles of their culture. AWS has its 16 leadership principles, which champion accountability, risk-taking, and the development of talent. Google defines a 10-point business philosophy, encompassing team achievements and pride in individual accomplishments that contribute to overall success. Microsoft talks about five cultural attributes, starting with its growth mindset: "always learning and insatiably curious." VMware has built its culture on execution, passion, integrity, customers and community (EPIC2). All of these are examples of generative cultures in action.

Teams over individuals

In *The Phoenix Project*, an allegorical novel set in a DevOps developer's nightmare, we meet a superhero developer named Brent. Every organization has a Brent. The person that knows how to do everything, with exceptional focus and problem-solving skills, and is helpful to everyone.

In a generative culture, this kind of superhero is a problem. Very talented superheroes are often overwhelmed with requests and meetings, predisposing them to become a bottleneck. As they do not scale, they first burn bright and then burn out. They make themselves irreplaceable but do not scale and leave anyway. Typically, they appear at the last minute, find a pragmatic solution to save the day, and then leave almost immediately afterward. This is symptomatic of a dysfunctional team culture.

Instead, cloud native cultures rely on learning organizations and balanced teams—autonomous groups of people with a variety of skills and perspectives that support each other toward a shared goal. They have the resources and authority to complete projects and are self-organizing, learning from each other through cross-disciplinary collaboration and iterative delivery. Through pairing and knowledge transfer, each team member could save the world.

Of course, nurturing a cloud native team does not just happen. It requires a foundation stone and strong dedication.

Flourishing in psychological safety

According to internal research by Google, psychological safety is the most important characteristic of high-performing teams. It allows team members to feel secure about taking risks. In such environments, people are better able to be themselves, make suggestions, and be vulnerable.

Crucially, psychological safety gives teams permission to learn by failing. Obviously, the goal is not to try to make mistakes but to accept error as part of the realm of possibility and use it to learn and improve. In other words, it's about allowing experimentation and play.

Play, learn, grow

An experiment is something we conduct to formulate and validate hypotheses. We plan just enough. We use lean product management and craft user tests. We escape analysis paralysis and explore fields of possibilities. Play loosens boundaries even further, while skyrocketing team motivation and commitment. It could take the form of chaos engineering featuring Netflix's monkeys, a capture the flag security game, a wheel of misfortune disaster role-play exercise, or some ChatOps bot. The playful names of these practices emphasize the generative nature of these processes: Although fun is not the point, the feeling of safety that comes with play is.

Adopt proven methods: Back to the future

At this point, you might feel overwhelmed by notions, ideas, and principles. Where should we best start? What is nice to have and what will make a major impact? Let's cover concrete practices and approaches to help transform the ways your application teams work into the ones of a fast-paced start-up, raising the development approach of legacy apps to great practices leveraged for brand-new app development. Interestingly, many of the opinionated techniques we'll review originated in the '90s and are still state of the art.

Articulate and share your foundational principles

Start exploring, brainstorming, and communicating on the foundations of your working mantras—how you collaborate, interact,and celebrate success. Do you empower teams, embrace change, deliver early and often, give back, and improve continuously? How important are empathy, kindness, and respect in all interactions? Very concretely, how does your ideal engineering culture look? Great sources of inspiration are the manifesto for agile software development, VMware's EPIC2 values, Apple's mission statement, Google's core values, Microsoft's corporate values, and Amazon's leadership principles.

Reflecting on this will help to evolve mindsets for the better. Including employees in the discussion is key to the adoption of your values. Those principles are likely to evolve over time after a major business change (see the introduction of Meta), rebranding (see Red Hat's new brand), merger, or acquisition.

Think big, start small, scale fast

Traditional modernization approaches reflect a waterfall approach. First, a long analysis phase dissects the current running applications to gain some insights on how bad the situation is. Then the target desired state is shaped, and the ideal target meta architecture is exhaustively specified before a transformation plan can be elaborated. This process could take so long that changes occurring in parallel could force a restart of most of the work. It could end in a situation similar to the Forth Bridge in Scotland that required continuous repainting. It's so long that once the painters reached one end, they had to start again at the other one.

To dodge and get out of this looping deadlock (AKA analysis paralysis), a more agile approach is crucial. Analysis and recurring tasks should be automated everywhere possible to get rid of tedious manual work and errors. Teams should plan just enough to start and begin with one thing. In practice, it could feel like letting engineering teams make a leap of faith. Adopting those highly focused and minimalistic mantras will swiftly reveal their value: quick feedback and progress, mitigated risks, strong alignment, and a solid basis to start scaling, letting the real work inform strategy. This is especially true for the most complicated tasks, such as the modernization of the aged monolithic systems covering the entire value chain of what your company does.

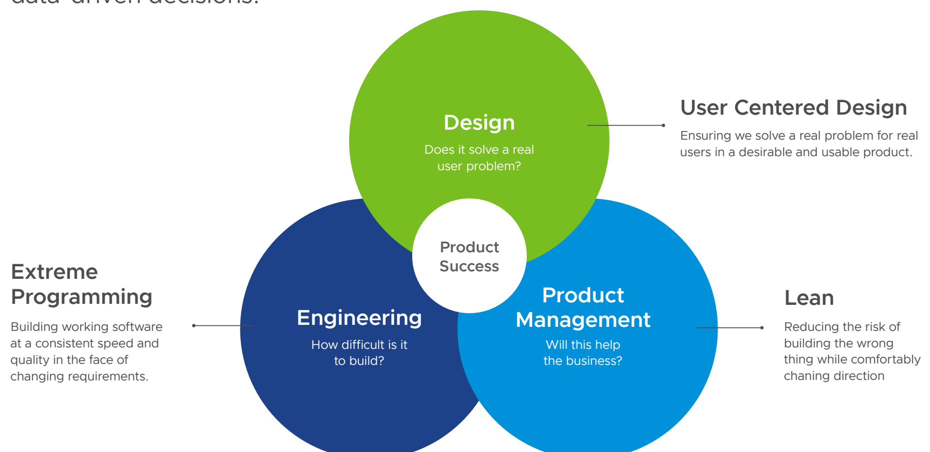
Adopt the fantastic three: UCD, XP, and lean

Three methodological ingredients from the agile software development world are instrumental to quickly incubating and developing new software products: extreme programming (XP), lean product management, and user-centered design (UCD). These methods are outstandingly impactful while modernizing existing applications, evolving them into a modern software product.

Extreme Programming (XP) is a set of software engineering practices targeting high quality, speed, and responsiveness to changing requirements. For example, it includes practices such as test-driven development (TDD), merciless refactoring, short iterations, CI/CD, and pair programming.

User-Centered Design (UCD) puts users—your customers—and their needs at the center of product design and development. It takes a data-driven approach to ensure the software you build solves real problems and delights your customers. Beyond user interface design, it focuses holistically on user experience while conducting user research, ethnographic studies, defining personas, producing prototypes, integrating accessibility, and practicing service design.

Lean product management strives to find the smartest and quickest way to build an impact-driven product that delivers value to customers in line with defined business outcomes. Lean uses constantly validated learnings to reduce the risk of building the wrong product while still being able to comfortably change direction. It includes a variety of practices that, for example, define a minimum viable product (MVP), conduct lean experiments, identify and test assumptions, and make data-driven decisions.



Together, these three methods get us closer to software excellence and success. To achieve this success, technology, code, tools, and frameworks only play a secondary role. The focus is on new things your customers care about and cherish, not being complete. Great products are desirable to users, viable to the business, and feasible for the engineers. Largely adopted to create new software products from scratch by silicon valley startups and large software companies, we will see how you can apply those methods to existing applications in the next chapter.

The new team

Typically, you recraft your teams to be cross-functional across development, design, and product management—balanced teams, as we call them. These teams take on a great deal of ownership and are given autonomy to make fast, very well-informed decisions and move quickly.

Growing collective knowledge with a cookbook

Many people also share practices by developing a cookbook customized to their organization. Often done as a wiki or some other easily updated document, these modernization cookbooks are collaboratively developed by engineers for their peers and comprise a growing set of proven recipes. A recipe is a short article explaining how to solve a specific problem or providing step-by-step instructions for completing a task. Recipes can include code snippets, patterns, or best practices. They're instrumental in sharing knowledge so that people on new projects avoid solving the same problem twice. Setup properly, we have seen cookbooks used prior to any Google and StackOverflow search to find solution articles. While maintained actively, such a knowledge base turns into the backbone of a modernization strategy at scale.

Internal platform

In parallel to the modernization efforts on the software engineering side, you should build a new home for your applications. An application platform is not an off-the-shelf piece of software; it's an evolving set of reusable services integrated with your existing systems that makes your developer more productive. Its capabilities should change in response to the needs of its users—your app developers—among whom it's a recognizable internal brand. In other words, your platform should be treated as a product and elaborated by a dedicated team.

Next up

Now that we covered the core principles to escape the modernization loop, let's put this all into practice in the context of rewriting existing monolithic systems from scratch.

Jump-start monolithic modernization

Thus far, we've discussed many of the tools for putting together your modernization strategy and plans. In this chapter, we're going to discuss Swift, the methodology we've been using for many years to help organizations modernize their oldest and largest applications and developed by Shaun Anderson. Swift is a comprehensive but quick approach to transforming monolithic applications into more modular, agile applications. As the name implies, the primary driver of Swift is to get started quickly.

Consider the monolith

First, let's define "monolith" in the context of software. This term is often pejorative[13] and used to refer to existing systems that are business critical, take a long time to update, are often expensive to run, and are technically complex. In other words, changing the software is slow and difficult. And when your business depends on software to run and compete, this means your business cannot easily change and adapt. You're stuck in the legacy trap.

Just one monolithic application is certainly difficult. But when you're dealing with multiple applications and services that are tightly coupled to each other, things get much worse. These systems of systems too often have brittle and inflexible architectures, are barely documented, and are sometimes a complete mystery to your current staff. Monolithic applications have many characteristics, but like so many large, troublesome things in life, you'll probably be able to smell the monolith right away. Let's look at an example of just that.

Case study: Trapped in a mainframe

One large organization we worked with made three unsuccessful attempts to decompose and modernize its mainframe application over the past decade. This application is one of the core services for the organization and is used daily by more than 3 million users and 35,000 administrative agents. Maintaining the monolithic application costs more than eight figures in a year. It's a 30-year-old mainframe running on IBM z/OS, with tens of thousands of programs written in COBOL that also used IBM DB2 and IMS databases.

Making changes to the application was costly. Releasing new or modified features into production took anywhere from 3 to 18 months. In comparison, more than 60 percent of organizations in a recent Cloud Native Computing Foundation survey release their software weekly, if not daily.

Like many other organizations, this one has been struggling to find a balance between making changes to comply with recently enacted laws and modernizing the technology used for its core systems. The organization also needed to bring the technology into the modern era: flexible cloud native architecture, modern programming language, simplified code, accelerated application development lifecycles, and lower costs. Because of the lack of thorough documentation, and with many of the organization's COBOL developers either retired or soon to retire, in-house domain and technical knowledge was sparse. This organization was stuck in a legacy trap, and anyone could smell it from a kilometer away[14].

---

13. Monolithic architectures can be a perfectly fine choice for the business and technical needs at hand. However, this term is often used as the enemy in application modernization discussions, so we'll do so here.

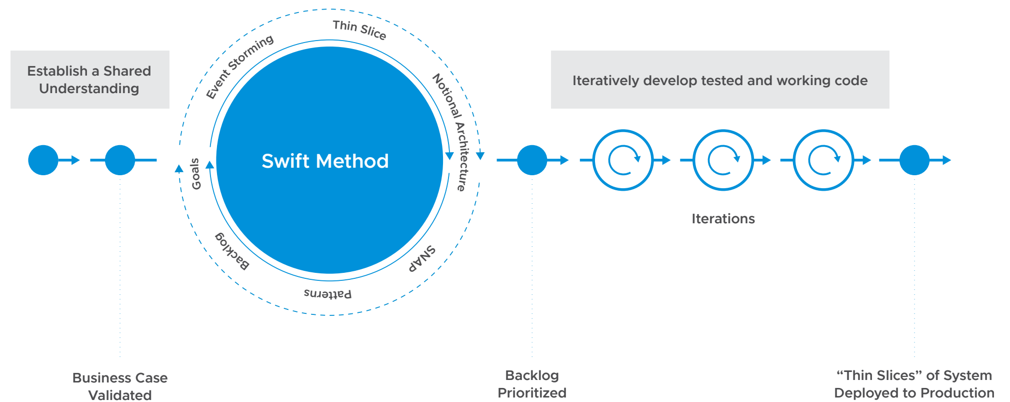
14. Or, 0.621371 miles, if you prefer.

The Swift method

True impactful modernization is not simply rewriting an existing application by leveraging new technologies. It's about understanding the system and how it holistically relates to the core business. In our experience, tackling large problems is best done incrementally, in small batches, giving you the chance to learn and adapt each time. One of the creators of extreme programming, Kent Beck, names this a succession:

"[The] art of taking a single conceptual change, breaking it into safe steps, and then finding an order for those steps that optimizes safety, feedback, and efficiency."

The Swift method takes this approach. Let's take a look at it.



The Swift modernization method was developed by Shaun Anderson within VMware Tanzu Labs™. It's unrelated to the eponymous programming language, framework, and payment network. The Swift method is a suite of practices building on each other. Executed in 4 to 10 short cycles, Swift jump-starts the work of turning an idea or a monolith application into a highly distributed, modern application. It helps bridge the gap in understanding between the nontechnical, top-down way of thinking and the technical, bottom-up thought process.

In the case of a legacy system, you decompose and rewrite your monolith from the ground up by figuring out how it wants to behave. At the antipode of a big-bang approach, Swift focuses you on a gradual transition. Old and new systems coexist while the legacy system is progressively broken down and its parts rebuilt. Following this incremental approach allows you to learn as you go and also gets you started sooner.

1. Swift is composed of six activities, done in the following order:
2. Quantify targeted modernization outcomes (1–2 hours)
3. Quickly understand the desired business functionality (1–2 days)
4. Select meaningful implementation starting points (2 hours)
5. Discover how the system wants to be designed (2–4 days)
6. Derive a prioritized backlog of work (1–2 days)
7. Craft iteratively tested working code (a few weeks)

As the times for each activity show, moving swiftly is built into the system.

Next, let's look at each of these activities in detail.

Quantify desired modernization outcomes

Finding and quantifying concrete objectives for your modernization strategy is the recommended first step of your modernization journey. What goals do you hope to achieve, and how will you know if you did? These goals are going to become the north star for your project or program. They'll help align new architecture and efforts with desired business outcomes.

To discover your goals and quantify the business outcomes, gather the stakeholders together and do the following:

1. Brainstorm concrete objectives for the upcoming weeks of modernization efforts. Like most goal-driven metrics, objectives should be ambitious and qualitative. Typical objectives could be to improve app modularity, automate app lifecycle, learn modernization practices, or validate high-level design.
2. Group similar objectives, forming clusters.
3. Prioritize the objectives (for example, by dot-voting on the objective clusters). Each participant gets three dots, representing votes to spend on the objectives they think are the most important. A prioritization between objectives will emerge.
4. Brainstorm key results for each of the top selected objectives that would help to validate if the related objective is fulfilled. Key results are quantitative and measurable. For example, support 10 times more requests on the rewritten login service.

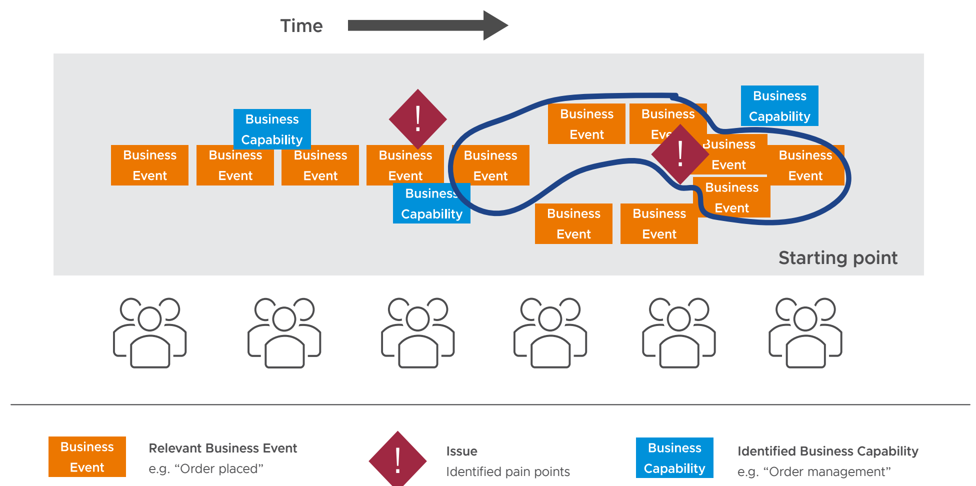
After a very intense one- to two-hour meeting, you will end with a prioritized list of agreed objectives reflecting the perspectives of all participants, supported by key measures to track them. These goals can be represented formally as OKRs, or whatever method you find most useful. The more important part is to find and agree on goals, and put in some thinking about how you'd gain success for each goal.

Quickly understand the desired business functionality

The aim of exploring the business domain behind your application is to gain a common understanding of what it does by building a simple, nontechnical, comprehensive model. It will foster alignment and form a ubiquitous language for all key stakeholders of the project: business analysts, software engineers, architects, managers, and others. Getting an overall picture of what the application does is essential to bring clarity to the desired target state.

Event storming is an incredibly efficient and fun method to capture and model the business behind any application. It comes from the domain-driven design (DDD) world. You can think of DDD as a collection of software development practices to find the common ground between business think and software design.

Invented by Alberto Brandolini in 2013, event storming is a workshop-based practice to bring together software developers and domain experts, and learn from each other. The exercise is extremely lightweight and intentionally requires only an experienced facilitator, a large wall, a few pens, and a decent stock of stickies. Although initially meant to be conducted in a physical room, it can also be performed remotely.



During a simplified "big picture" variant of an event-storming workshop, all key stakeholders of your monolithic application are invited to participate: architects, business leads, software developers, management, operation specialists… Collectively, they storm out a series of business events triggered within the future modernized application. Orange sticky notes are used to represent business events. In several iterations guided by a facilitator, the group will collaboratively let the model emerge and sort events chronologically from the left to the right of the wall.

Then, events belonging to the same business domain are grouped together in a business capability or business domain. This way, the participants will jointly identify the distinct and independent business capabilities provided by your monolith.

While being intense, highly interactive, and a bit chaotic, the exercise yields an extensive model of your modernized monolith—a wall covered by clustered stickies—within only one to two days. It enables cross-perspective conversations while projecting and consolidating head knowledge on one single wall.This workshop is highly effective. We have seen highly complex applications modeled in two days that months of documentation efforts failed to grasp.

Find and prioritize meaningful implementation starting points

Once you gain a crisp understanding of what your monolith is doing, you need to figure out where to start. For example, we've supported a large financial software vendor wrangling to find a path forward after multiple generic event-storming exercises conducted on the same application. The team was struggling to validate its meta model and select the first business narrative to reimplement.

This practice identifies the first business narrative (or workflow) to start modernizing out of everything your monolithic application is doing. We call this business narrative a thin slice of functionality as it reflects a small functional piece of the whole cake—your monolith application.

First, it starts with a brainstorming exercise to identify the biggest issues and challenges, or the biggest chances and opportunities within the event-stormed application. Those are directly captured as stickies in a different color (usually red) and stuck close to the business events they're related to. Subsequently, the created issues are reviewed, discussed and consolidated by the group.

Then, the biggest issues are prioritized through dot-voting. Looking back at the previously captured goals before opening the vote sometimes helps. The events related to the issues with the highest votes are great to include in the first thin slice.

Informed by those insights, one group of business events (orange stickies) covering a clear business narrative is defined. A good rule of thumb is to pick a not too simple but happy path with some of the identified issues, without touching everything. The selected thin slice will form the first piece of your monolith to rewrite. The exercise should not take more than two hours.

Discover how the system wants to be designed

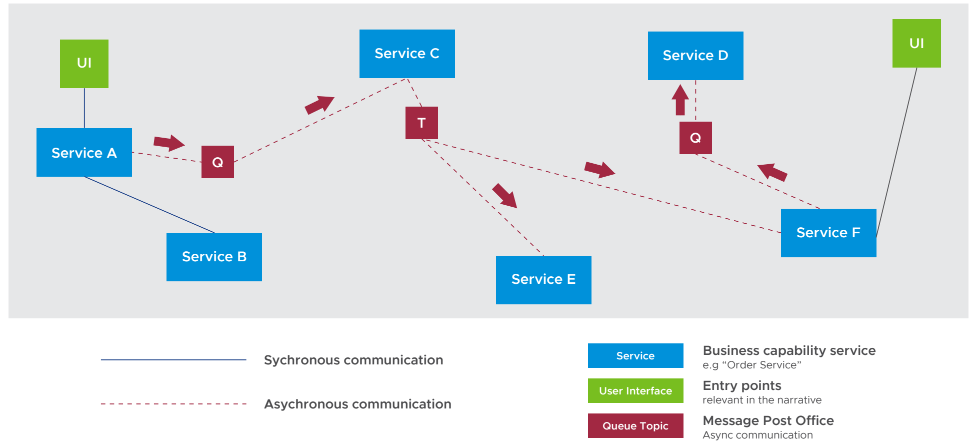
So far, you have identified a clear business narrative as a starting point for your modernization efforts. How do you turn this into code? Similar to a house renovation project, an intermediary step in crafting the high-level building plans—also called notional architecture—is as necessary as useful to have all stakeholders aligned.

At this point, it might be tempting to specify every bit of the use case in detail, drafting the perfect high-level system design including its connected meta model and architecture. Such a big design up front (or waterfall) approach is likely to take months. This too often makes the big design outdated by the time the system is completed. Perfect is the enemy of good.

Within just a few days, we want to consolidate the design skeleton of our application. This process generates information about how the system wants to be designed and attempts to avoid pitfalls such as premature solutioning. It achieves a quick consensus on the desired high-level design, revealing APIs, services, data, and event choreography.

A pragmatic approach to figuring out how the system wants to behave starts by modeling the communication between the business capabilities identified during the event-storming exercise. This is what the "Boris" exercise is all about. Each business capability and each user interface are represented by one sticky note on a physical or digital wall.

Led by a facilitator, the team goes through the selected thin slice narrative to model the communication between the identified business-centric services and user interface. Each communication is represented by a directed line between stickies. It can occur synchronously, having the caller wait for an answer, or asynchronously passing messages to a broker.

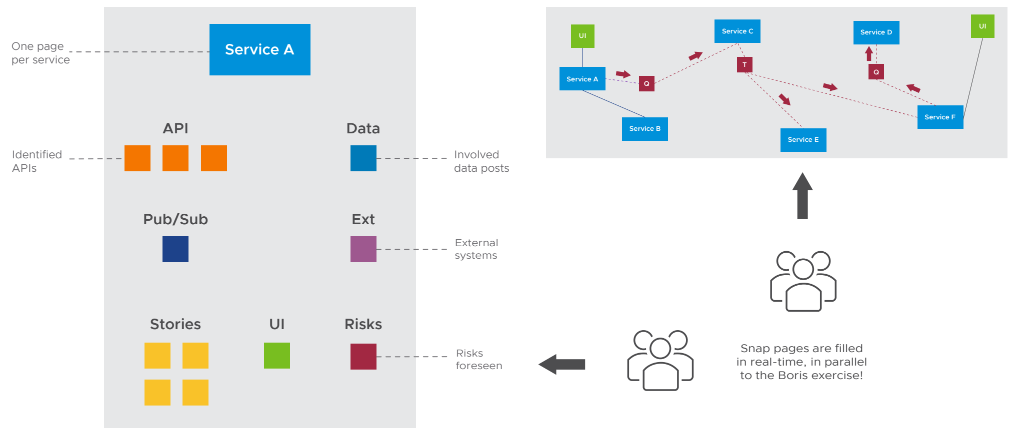


At the end of the exercise, you obtain an annotated diagram that looks like a spider web meticulously describing all communication flows covering your business narrative. This step typically takes twice as long as the event-storming exercise.

Derive a prioritized backlog of work

For each identified service, we want to craft actionable user stories to start their implementation. Generating this backlog of work will help move to the technical design and implementation phase.

The snap technique quickly documents the outcomes of the Boris exercise in real time. While conducting the Boris exercise, you use one page per service to document all relevant aspects of the involved data pots, identified APIs, message brokers (queues and topics), dedicated user stories, linked user interfaces, foreseen risks, and connected external systems.



A few dedicated team members focus on capturing all those elements while the team is moving through the selected business scenario.

Once the interactive Boris exercise is completed, all captured information is ready to be translated into a backlog of work. This is typically orchestrated by a product manager supported by an experienced software architect. Together, they formulate the stories and add the required chores and spikes.

Chores are activities teams perform to do work or to work more easily. Spikes are stories for which the team cannot estimate the effort needed. They can be seen as time-boxed, exploratory research work to learn about the issue or the possible solutions.

Iteratively craft tested and working code

From there onward, the development team follows a classical agile project model. They iterate on the backlog, work in sprints, test and demonstrate their achievements, and conduct a retrospective before they loop into the next sprint planning.

The first stories the team works on are often architectural spikes exploring technical design patterns, making technological decisions, and studying ways to bring together the existing and new worlds. Initial chores include the full automation of the path to production, paving the way for a reliable, zero-downtime deployment and software promotion process.

A few iterations will be necessary to bring the first modernized service to production. Many more will be required to completely replace the existing legacy system.

Back to our customer

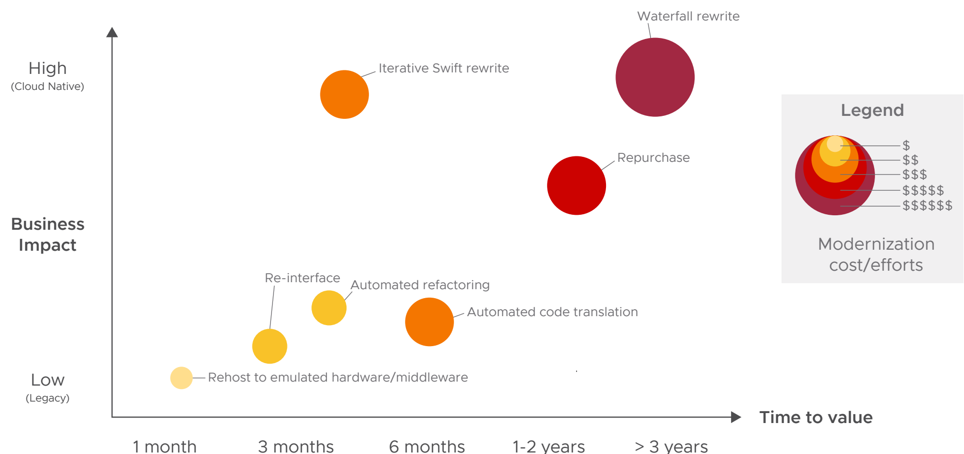
Exactly by following those steps, the previously mentioned, mainframe-modernizing organization found a way out of the legacy trap. Beyond successfully initiating the rewrite of their mainframe, the team managed to bring the domain and technical knowledge about their monolith back internally and seed a startup-like software development culture. Leveraging pair programming between software engineers, they progressively scaled by sharing and spreading those new practices among a wider group of developers.

After nine weeks of work, the first rewritten service was deployed to production with the ability to progressively transfer traffic between old and new worlds without the end users noticing as the software was modernized. Last but not least, the whole path to production has been automated to deploy code changes without any downtime, within 30 minutes to production instead of four weeks in the past.

Common missteps with traditional approaches

Now that we know the right way to approach large-scale modernization, let’s take a look at some of the missteps organizations often make when modernizing monolithic applications. By going through each anti-pattern and explaining what they’re about and why they’re not effective, you can avoid falling into the legacy trap.

The following diagram depicts the traditional modernization approaches for monolithic applications we will cover and compares them to Swift. It sets side by side each approach in terms of typical time to realize value in production, transformation cost and efforts, resulting business impact and how modern the target application ranging from “legacy” to “cloud native.”



Rehost to emulated hardware/middleware

What?

Software is rehosted from an expensive hardware or middleware layer to commodity hardware, emulated infrastructure, or lower-cost, cloud-compatible technologies.

Why not?

The trap with this approach is to focus too much and too narrowly on only a few costs. Despite much analysis, we’ve seen many rehosting strategies fail to take into account the cost of everything that’s involved in the change, not to mention second order effects. For example, the new middleware and infrastructure you use might be much cheaper, but the amount of time and money it takes to retrain your staff could erase savings. This is a common perception of open source software which, though free, is often not polished enough for the amount of time and money enterprises want to spend on the software. Another trap is ignoring the costs of validating that you have the same functionality with an improved nonfunctional context. That is, making sure everything still works, including the enterprise’s services and apps that you have not modernized or have control over. Expect these deeper pains to resurface in the middle term: lack of in-house domain knowledge and skills for older programming languages (COBOL, RPG, PL, PowerBuilder, etc.) and legacy middleware in use (older Java application servers), limiting architecture, inflexible time-consuming processes.

There are more favorable reasons to rehost, sometimes also driven by the need to gain more security and performance capabilities than your current data centers provide. These cases are usually not primarily focused on lower costs but on pure need. Running MS-DOS or Windows 95 in the cloud won’t change its architecture to make it more scalable, modular or secure.

Automated code translation

What?

Tools are used to automatically convert application code to a more modern programming language, such as Java, C#, JavaScript, Go, or Rust.

Why not?

Each generation of programming languages introduces new concepts, methods of design, and the associated tooling needed to write and support that code. For example, the move from procedural code to object-oriented code changed everything. Each generation of programming languages thinks differently than the others. The benefits of new languages (recently, for example, the ability to handle web-scale amounts of data and transactions) are often lost when code is auto-generated. In addition to that, the generated code ends up difficult to read, maintain, and extend, and loses touch with its existing code version history and documentation. The impact of any translation error could end up being massive with programs having millions of lines of code. Last but not least, such a translation won't provide a more adapted application architecture fitting your needs for scale, reliability, modularity, and the like. As an analogy, imagine you were translating a manual for horse-drawn carriages from French to English, but your new business need was to manage driverless 18 wheelers. The words could be translated, perhaps even poetically, but they would be useless for your new needs. As with rehosting, you'll also encounter unplanned costs and time when it comes to testing and validation.

Automated refactoring

What?

Widely known refactoring activities are automated to clean up and restructure code to follow best practices. Update libraries, or put new types of software design in place. This automated refactoring is analogous to text entry fields on most of today's devices that auto-correct misspellings and suggest grammar fixes.

Why not?

Code is complex—there are many ways to write even the simplest of operations. There are enough common patterns to automate rewriting of large pieces of code. However, automated code refactoring must be verified, and it's the ability to verify changes to legacy code with tests that's often the problem. Furthermore, automated code refactoring will not change or update the business logic in your applications; that is what the code does[15]. The promise of a tool automating code creation and refactoring toward microservices and clean code is fulfilled only on a very narrow set of use cases today. Tools from Snyk, GitHub Copilot, vFunction, Spring, and many others look promising, especially as they bring determinism to the conducted changes.

---

15. A purist would argue that, by definition, refactoring should not change the outcomes of code. Instead, refactoring is intended to improve how the code functions, not what the code does.

Code refactoring will only take you so far. It can be useful, but it's only part of escaping the legacy trap. When it comes to understanding the business domain of your application, in order to architect an appropriate high-level design aligned against desired business outcomes, tools alone won't help much. They remain tools and will neither upgrade team culture and nor ways of working.

Re-interface

What?

Create a new app exposing functionality of your legacy system and put it in front of it. This is commonly known as a facade pattern and is a tactic to mask complexity and foster the reuse of existing functionality.

Why not?

While combined with a strangler pattern, facades provide good support to a longer modernization projects breaking down your monolith bit by bit. However, this approach is not the endgame. This pattern does not really touches or modernize your monolith. It just hides the technical misery and tightens system coupling.

Such a facade increases the performance pressure on the existing system by exposing it to new clients, without improving its core design. Worse, it adds more dependencies to your already tightly coupled monolith. We've seen many similar projects that first introduce a small facade, before evolving into an API gateway and then into a fully fledged dedicated API management platform with advanced access management and observability capabilities. If neglected, this new layer can just turn into something new to worry about, increasing the complexity of the initial distributed monolith's universe.

Repurchase

What?

An equivalent COTS software is bought to replace your legacy app.

Why not?

This sounds as straightforward as signing a big check and pressing a metaphoric migrate button. For very standard use cases that have migration scenarios fully covered by the target vendor, it might be the case.

However, for bespoke software covering more complex business logic, it's likely to come up short and end in a longer-than-expected waterfall project. These are the most common reasons the button push doesn't work:

- Your customizations aren't easily translatable or simply aren't possible with the new system. In the worst case, this requires as much code writing as in a full rewrite from scratch.
- Some data must be manually transformed instead of transformed with standard automation. Data conversion, export, import, transfer, and validation could drain a lot of time and efforts.
- Functionality mismatches between the old and new system require manual intervention.

Often, introducing an off-the-shelf application will also change the workflows in the app, meaning people will need to relearn the new system. While learning the new app, productivity may fall and people may often miss old features that no longer exist in the app.

Waterfall rewrite

What?

A waterfall rewrite project structures the replacement of an existing monolith following the V-model for software development. The V-model comprises a multi-month design phase covering requirement analysis, system design, architectural design, and module design, as well as matching to an ideal metadata model. Developers then write the new application code. Each stage is followed by a thorough validation check.

Why not?

Projects following this approach often get stuck in what armchair psychologists call analysis paralysis. Having too much data can hinder the accuracy of decisions or the speed with which they're made. The sheer volume of available information on the existing application—decades of documentation, code base, technology clutter, competitive market collateral, company strategy, differentiating ideas and aspirations—makes it difficult to get a clear picture and define an optimal target state.

This modernization endeavor is so massive that its design phase feels like the never-ending task of painting the Forth Bridge in Scotland. In that instance, once completed, the fundamental premises of the design study changed so much that it could be restarted from scratch.

In a nutshell, although a rewriting approach is the best way to radically improve your applications, it cannot be conducted in a waterfall manner. As stated by Albert Einstein, it leads us to realize that "we cannot solve our problems with the same thinking we used when we created them." In software, if you're writing a new application from scratch, it's a good idea to take advantage of the lean product management approach to development while proceeding in short iterations. This practice focuses on verifying your assumptions with each release rather than assuming they're all correct from the beginning. For more discussion on that approach, see Monolithic Transformation.

It always depends

We've pointed out the flaws, but there are techniques in each of these approaches that are helpful. However, they're generally helpful when used together with other techniques rather than in isolation.

Building a modernization habit

Now we'll tell you a discouraging truth: You will never finish modernizing your application portfolio. A modernized app today is just a legacy app tomorrow. (Well, 5 or 10 years from now.) To escape the legacy trap, application modernization must become an ongoing habit. The sheer amount of applications most large organizations have—in the thousands—means you could spend years modernizing your portfolio. Hopefully, you can rehost, replace, or retire many of those, saving much time and money. However, your software estate needs constant maintenance just as a city needs constant maintenance to remain livable.

If you've fallen into the legacy trap, it means you haven't built up the application modernization habit. It hasn't been enshrined in your IT culture. After learning how modernization works in your organization, hopefully with proven methods to prioritize and align modernization efforts and modernize monolith apps (Swift), start thinking about how you can install it as a permanent part of your organization's process and culture.

We work with too many organizations held back by the legacy trap. And worse, we have to use applications from many of those organizations, as you likely do. Modernizing your application portfolio will certainly make your organization run better, but it will also likely improve people's day-to-day lives.

Just remember to guide your strategy by business need, start small, follow a disciplined process, and learn each cycle. There are many tools to help modernize your applications, but the real secret is to actually make sure you and your organization are doing the work week to week.

Good luck!

About the Authors

Michael Coté studies how large organizations get better at building software to run better and grow their business. His books *Changing Mindsets*, *Monolithic Transformation,* and The *Business Bottleneck* cover this topic. He's been an industry analyst at RedMonk and 451 Research, done corporate strategy and M&A, and was a programmer. He also co-hosts several podcasts, including Software Defined Talk. Cf. cote.io, and is @cote in Twitter.

Marc Zottner spends most of his time supporting large companies to modernize their strategic applications. At VMware Tanzu Labs, he is the global application modernization lead. Former transformation program lead, consulting architect, middleware specialist, and trainer, Marc has extensive experience in the application ecosystem and often feels like an archeologist.

This book was compiled in October 2022 by VMware.

Thanks for reading!

vmware®

Copyright © 2022 VMware, Inc. All rights reserved. VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001
VMware and the VMware logo are registered trademarks or trademarks of VMware, Inc. and its subsidiaries in the United States and other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies. VMware products are covered by one or more patents listed at vmware.com/go/patents.
Item No: The Legacy Trap 10/22