GlobalLogic®

The Economics of DIGITAL Transformation

Dr. Jim Walsh
CTO, GlobalLogic

GlobalLogic

Contents

3 Introduction
9 *What is Digital Transformation, and Why is it Hard?*
13 *Software-Enabled Business*
18 *Time and Cost Profiles*

22 The Greenfield Approach
26 *Greenfield Case Study*
36 *Greenfield Pros and Cons*

43 The Side-by-Side Approach
49 *Integration at the Glass*
51 *Integration at the Glass Case Study*
62 *Extend and Replace*
64 *Extend and Replace Case Study*
68 *Present-Forward / Future-Back*
73 *Side-by-Side Pros and Cons*

74 The Gradual Evolution Approach
81 *Gradual Evolution Case Study*
92 *Gradual Evolution Pros and Cons*

96 The ROI of Digital Transformation
99 *Transformation Approaches*
100 *ROI and Financial Impact*
104 *Analysis*

112 Why Do Digital Transformations Fail?

119 Summary and Conclusions

Introduction

Introduction

There are a number of paths to digital transformation, but they all start where you are today. Which path you choose depends on where you are, both internally and externally, and on the economics and risk tolerance of your company.

When discussing economics, minds often turn to finance. As the popular "Freakonomics" (by Levitt & Dubner) series of books and podcasts have made us aware, economic thinking covers a broader range of topics than finance. In fact, as we shall see in this paper, the most important financial aspect of digital transformation is its ability to position your company or line of business for future success and to avoid literal failure of a mission or enterprise. While we will discuss the ROI of digital transformation projects in the final section of this paper, ROI and other financial metrics are not the focus of these essays.

Instead, we will use the word "economics" in its broader sense—that is, we will talk about making wise choices to allocate finite resources in order to achieve a desired future goal in the presence of competing priorities and constraints. The "resources" in this picture do not include money alone, but also time, effort, management and market attention, and many other factors.

If you are reading this, you have probably already determined your future will include digital, which means you are now choosing how to execute a digital transformation. In this activity, you will be migrating your company or line of business from its current state to a place where it increasingly engages employees, customers, partners, and other actors (human or otherwise) through digital as opposed to traditional channels. While this paper will primarily focus on "How" to execute a digital transformation operationally, we address the "What" and the "Why" in the next section, as well as those at the end of the piece.

We will also discuss the economics and other considerations that drive the selection of an operational and technical approach to digital transformation. There are also a number of non-technical factors which operate in parallel and can affect the cost and time profile of your transformation. These non-technical factors include staffing and training costs, organization, culture, appetite for process change and adoption, product management and design capabilities,

time-to-market considerations, board and investor commitments, macroeconomic conditions, your competitive landscape, and many other factors. Many of these will be touched on in the course of these discussions, as seen through an operational and technical / engineering lens.

Additionally, we will focus on companies with existing systems, existing teams, and existing ways of working. While we enjoy our work with startups and "1.0" products, the transformation situations we encounter most often are sizable (multi-$100M to multi-$10B USD revenue) business units and companies who have significant-scale existing systems. If you are in a situation where you need or choose to start fresh, your approach is almost determined for you: Greenfield. Even within true Greenfield opportunities, however, many of the same economic and other factors come into play, so we hope you'll find this piece of value in your situation as well.

After leading or participating in many large-scale digital transformation projects, GlobalLogic has identified three primary technical transformation patterns that are widely applicable:

- Greenfield: A new system is created "from scratch" without any mandate to reuse an existing system or code. Data, content, and customers are migrated to the new system when it is ready, and the old system(s) it replaces is decommissioned.

- Side-by-Side: In this approach, a partial implementation of a modern Greenfield Next Generation system is created and deployed while the "legacy" system is still in production. The NextGen system either (a) implements and replaces part of the legacy system's functionality, or (b) extends the legacy system by providing new capabilities.

  From a feature delivery standpoint, during the implementation of a Side-by-Side approach, some system functionality is provided to end users by the legacy system, and other functionality by NextGen. Often this bifurcation is hidden from the users, either by making the NextGen system backward compatible from an external interaction perspective, or by upgrading the legacy system to support a new interaction paradigm introduced as part of NextGen. As part of a transformation scenario, feature delivery will be migrated, over time, off the legacy system and re-implemented in the NextGen system, allowing all or part of the legacy system(s) to be retired.

There are several variants of the Side-by-Side approach that we will discuss:

"Integration at the Glass": In this variant, the old and new systems are deployed in production simultaneously, with their only real touchpoint being the experience layer. The "glass" referred to here is a computer screen, but the same principle applies for any interface point. In this approach, the "old" and "new" systems only appear to be integrated; in fact, they exist side-by-side with little or no interaction between them. As a transformation strategy, the objective is to replace the old system with the new one over time, while at each point making it appear to end users that there is a single system "behind the glass." This approach is light-weight, relatively low overhead, and effective for some systems. However, it is not universally effective, and not easily sustainable indefinitely even for systems where it's a fit.

"Extend and Replace": In this variant, an explicit goal is to replace the existing legacy system while changing it as little as possible. The mandate to leave the existing system alone is often motivated by a concern that the legacy system is too fragile to be changed, a belief that the current system is too costly or not practical to change (because of lost skill sets, obsolete technologies, etc.), or because the stakeholders see no value in making any more investment than absolutely needed in modifying a legacy system that will ultimately be retired.

In this scenario, a partial "vertical slice" implementation of a modern Greenfield "Next Generation" system is created and integrated with the legacy in a "minimum destabilizing way". The legacy and partial NextGen systems are then deployed in production simultaneously, with the legacy system supplying some functionality and the NextGen system providing the rest. As the NextGen system becomes more capable over time, it progressively takes over more and more of the functionality of the legacy system until the entire legacy system has been replaced and can be retired.

"Present-Forward" / "Future-Back": In this approach, both the legacy system and a Greenfield system are simultaneously worked on, with the goal of creating a single, common, improved system architecture. The work is done in parallel, with one team refactoring and evolving elements of the current system (the "Present-Forward" team) while a second team

develops new or replacement components using a Greenfield paradigm (the "Future-back" initiative). The efforts of both teams are coordinated to "meet in the middle" in a next-generation system that is a hybrid of refactored elements of the current legacy system as well as freshly created Greenfield" components. This approach is most feasible when the target next-generation system is a heavily componentized architecture that supports polyglot (different language) components as well as multiple forms of persistence. Using a containerized microservices architecture as an end-state is one example of where this approach can work.

- "Wrap-and-Refactor" / "Gradual Evolution": The Gradual Evolution approach is the third major approach to digital transformation. In the Gradual Evolution approach, the existing system is gradually and continuously "morphed" until it becomes the desired NextGen system. Reuse of the existing code base and continuous operation of the current system are often key considerations driving adoption of this paradigm.

  "Wrap and refactor" is one of the technical methods used to achieve the Gradual Evolution into a next-generation system—in fact, it is so commonly used that it is basically synonymous with gradual evolution. In the "wrap and refactor" approach, a modular structure is (conceptually) overlaid on the current "as-built" system. Many legacy systems are not, in fact, truly modular; imposing a modular structure on them is initially an idealized mapping exercise rather than a real reflection of the as-built structure of the system.

  The actual system implementation is then made physically modular by "refactoring" (modifying) the code within the newly-defined logical component boundaries. The intent of this refactoring is to make the interactions between modules architecturally explicit through a well-defined interface (such as an API or event structure), while also orthogonalizing the implement code by removing implicit "side effects" of one module's implementation on another's. The removal of side-effects generally requires significant refactoring of code and also the way shared resources are handled—data stores in particular. In parallel with this refactoring, a next-generation "as desired" architecture is designed, and the now-modularized legacy system architecture is brought into compliance with it.

Many companies with complex systems use a combination of these approaches;for example, one subsystem might receive a Side-by-Side treatment while another is refactored. While these are the three primary approaches, the technical transformation situations we encounter in real life will - almost without exception - use variants of one or more of them.

There is no “always right” or “always wrong” approach to transforming a current system or set of systems into a new one; each approach has pros and cons. This is especially true on the cost side, since many companies are forced into digital transformation either by an imperative to reduce engineering operations and development costs, or to grow or protect revenues by introducing new (or neutralizing) features rapidly while increasing engineering costs modestly, if at all. All three of the basic transformation approaches have very different cost, risk, and time profiles. The “silver bullet” is choosing the approach that is most suited to a given system, company, and business situation.

In the following sections, we will discuss the pros and cons of these three fundamental approaches in more detail. But first, let’s back up and discuss what digital transformation is and why it can be so challenging.

What is Digital Transformation and Why is it Hard?

Historians will argue about the specifics of the digital revolution for many years to come, but let's try to frame what happened and why it matters in a way that's actionable. During the time period from roughly 1994 (when the Netscape Web Browser first shipped) through roughly 2008 (when Apple's AppStore opened and the Android smartphone first shipped), a set of technologies emerged that had the potential to profoundly change how people lived, worked, and interacted. Specifically, regular people could now connect with businesses and each other globally through the world-wide web and through "apps" hosted on smart mobile devices.

Concurrently with these new ways of human connection, a set of software tools and approaches were introduced that could deliver business value through those new web and mobile interactions at a mass scale. The new technologies coming to market in those 14 years included:

- The stateless REST interface paradigm (1994 / 2000)
- HTML/CSS/JavaScript web applications (mid-1990s)
- The public cloud (2006)
- Modern NoSQL (term coined in 1998)
- Virtualization of commodity hardware (VMware founded in 1998)
- Smart mobile devices like the iPhone (2007) and Android (2008) and their corresponding app development frameworks
- Modern downloadable mobile applications (Apple AppStore 2008)

Also, in the same timeframe, new low-cost / high-speed wireless data connection paradigms emerged that would untether users and become ubiquitous. These communication technologies included:

- Wi-Fi (Wi-Fi alliance founded in 1999)
- texting (SMS over GSM first deployed commercially in 1995)
- 3G (2001) and later 4G (2006) mobile broadband infrastructure.

These technologies had precursors as well as successors; however, the seeds of what we consider our modern world were clearly planted over a decade ago.

During that same time period, new companies began to take advantage of the new digital technologies and their capabilities while advancing those technologies and paradigms themselves. These "digital natives" included Amazon (1994), Google (1998), Facebook (2004), Twitter (2006), and many others. Some established companies also successfully transformed to become digital giants, with Apple being a conspicuous example.

While the period from 1994 to 2008 arguably laid the foundations for the digital revolution both technically and in daily life, many places are just beginning to feel the most profound effects. Like an earthquake that happens mid-ocean, it takes a while before the tsunami reaches the shore. Also, as in a tsunami, the effects intensify as the waves near the beach. Fed by increasing computational power and declining infrastructure costs, digital technologies and infrastructure have continued to advance.

Over the last decade, significantly more compute power per dollar has made it practical to deliver computationally expensive technologies such as Artificial Intelligence (AI), Machine Learning (ML), computer vision, speech recognition and many others at scale - affordably and in compact footprints - directly to consumers. Business models and consumer behavior have kept pace and continue to rapidly evolve as well.

The net effect is that we live in a business and cultural environment that has changed almost beyond recognition within a generation. We now inhabit a digital world. The digital transformation of a business is nothing more than a response to the digital transformation which has already taken place in the world around us. Businesses that are not digital are at risk, because they no longer live in the same world that their customers, employees, and, increasingly, their competitors now inhabit.

Why did it take ten years for this tsunami to reach the shore? The tendency in any business is to use a new technology to enable them to better, faster, or more cheaply accomplish the things they already do. In about 99.9% of the cases, this is exactly the right thing, because the overwhelming majority of technology changes

are incremental. The problem occurs when we continue this same pattern while the game has changed fundamentally; that is, when we treat a transformational technology as an incremental one. Transformational technologies do not happen very often, but when they do, we need to recognize and respond by transforming as well, or we risk becoming irrelevant. History is littered with those who either did not respond, or who responded too late to the fundamental transformation technologies of their time: the industrial revolution, electricity, the internal combustion engine, the computer. In our time the transformative technology is digital.

It has taken a decade for the transformational nature of the technologies introduced between 1994 and 2008 to become obvious, but it is now. Digital giants have profitable revenue streams in the tens to hundreds of billions of dollars annually. Over half the world's population is connected to the internet. A single company (Facebook) has more actively engaged users than the population of the most populous nations on Earth,and that company is growing much faster than any of those countries.

Once we admit there is a need to transform, what does it mean to actually do it? The word "digital" is used so often that it's lost all meaning. Let's back up and define it, as we will use it in this document.

In our terms, "digital" simply means "enabled by software and data". This is in contrast to "physical" systems where the "intelligence" behind an interaction is supplied by human beings, and the interactions themselves are governed solely by the rules of the physical world, such as mechanics, optics, chemistry and electricity. A customer shopping in a brick-and-mortar retail store and a person receiving a treatment from a massage therapist are two examples of business value provided through "physical" channels.

Shopping online using an app on your smartphone is an example of a "digital" interaction. Your interaction is mediated by software running on your smartphone (the app), as well as in the cloud and probably in the retailer's data center. While the connection between your phone and the retailer's computer is also "digital" in the sense that your phone sends and receives binary digits over its radio, this connection is not what makes an interaction "digital" in our sense of the word —it's the software behind it.

Definitions can get tricky in hybrid scenarios: For example, a customer shopping in a physical retail store but doing price comparisons on his or her phone while they are shopping: Are they having a "physical" experience or a "digital" one? Of course, the answer is "both," but for the purposes of this paper, the important point is who that shopper is interacting with. If he or she is engaging both digitally and physically with the same retailer, then that store is being successful as a digitally enabled entity. If he or she is interacting with a competitor on their phone, however, while taking advantage of another retailer's physical presence, then the physical retailer has not been successful as a digital business in this scenario.

A colleague put it well when he said that a business is "digital" to the extent that the things it relates to are "virtualized". Uber, Lyft, and Airbnb are frequently cited examples of "pure" digital businesses, and rightly so. None of these companies owns the things they sell—Uber and Lyft own no cars, and Airbnb owns no properties. Each of these companies operate by owning, managing and processing data about those things—not by owning the things themselves. From the perspective of the business, the cars and the homes they offer are "virtual"—all they are is data. In that sense, these companies are purely digital—all they are is software and data.

At this writing, the jury is still out concerning the success of 100% digital entities in the market. The barriers to financial success are huge, partly because the physical systems being displaced (whether inside or outside a company) tend to fight back.. Fortunately, not all businesses need to or should become purely digital entities, though, the thought process about how to do so is key to success in a digital age.

In our work with businesses, we identify three types of companies that depend on software as a fundamental aspect of their business. There are no universally agreed upon definitions of these terms, but we find the following ones useful in our work, and will employ them in the balance of this document:

- A digitally-enabled business is an organization that delivers business value and generates revenue directly through its use of one or more software-mediated channels. A digitally-enabled business not only presents itself through Web, mobile or other channels, it also uses these channels to directly engage with end-users and deliver value to them. Revenue may come directly from the end users or indirectly through customers who pay for access to those end users—for example, advertisers or merchants. In a digitally-enabled business, value is created through the use of digital channels, directly or indirectly.

- A “digital native” or “digitally transformed” business: To what extent are the business’s interactions with the physical world “virtualized”?

All software is based on a process of abstraction—software manages data about physical things, not the thing itself. Even where software is used to directly actuate physical devices—an industrial robot, a wind turbine, a door lock, a heart pacemaker—it is not the software itself that interacts with the world. Instead, the software set parameters that are consumed by a physical device. The physical device performs the action, not the software.

Software-Enabled Businesses

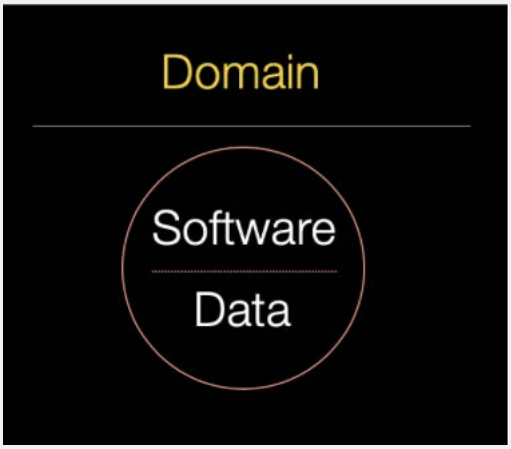

Figure 1: Traditional business

A software-enabled business is one whose primary business activity is *supported* by software programs running on a computer, phone, or other physical or virtual device. The software itself does not directly provide business value to customers of the business; the customer receives business value through the physical delivery of goods or services. Software is used by the employees of the business in order to do their jobs or functions such as tracking their work, creating invoices, or receiving payment. It is hard to imagine any late-20th or early 21st Century businesses that do not fall into this category at minimum. While such a business might have a website that advertises its offerings, they do not directly engage in the delivery of business value through the website itself or through other software-mediated channels.

Software-enabled businesses may also use digital channels in incidental ways, such as receiving payment for goods or services that are ordered, delivered, and invoiced through other means. They may also use communication services such as Web conferencing or Facetime conversations to conduct business. The key distinction, though, is that these tools are incidental to value creation by the business; the software itself is not what creates the business value.

A traditional retailer operating exclusively through "brick-and-mortar" stores is a good example of a software-enabled business. Even the most traditional retailer will generally be supported by a wide variety of software. These might include warehouse management systems, payroll, accounting, POS, spreadsheets, and many other systems. A services business, such as a Psychologist, is another example. They might use software to schedule appointments, track their hours, and file insurance claims. They might also allow their clients to pay online or use Facetime or Web conferencing to conduct remote sessions. However, the "business value" delivered by the Psychologist is the therapy session, not the software.

A traditional business will also have an IT department—or if they're really tiny, designated individuals—who keep their software systems up-and-running and augment or upgrade them periodically according to business needs.

The less money such a traditional business spends on supporting activities, including software and IT, the more profitable it can be (all things being equal). This is because the traditional business's revenue is generated by its "business domain" activities. For example, for a traditional retailer, this might be sales of items to consumers in a physical storefront. Supporting software and other IT activities are just expense items in a traditional company. Our figure is drawn to suggest a mathematical fraction, and like a fraction, a company following a traditional model makes more profits by growing the top-line revenue in the numerator, while shrinking its expenses for software in the denominator. In other words, success in a traditional company forces IT to be an efficiency play.

The number of completely traditional companies is decreasing year by year, of course, because the importance of digital has been clear for some time. Nearly every company has found some way to leverage digital to enhance their current revenue streams. You would be hard-pressed to find a brick-and-mortar retailer of any size without an active Web and mobile presence, for example. Does that make these companies digital?

Using digital to drive incremental revenue to your traditional revenue sources (e.g. sales of your retail items) is nearly always a good thing, but is not in itself a digital transformation. It's a channel strategy. So what is "true" digital?

Figure 2: Digitally transforming business

A business that fully embraces digital generates revenue through non-traditional sources enabled by the software itself. This is because "digital" makes new ways to interact with consumers and employees possible. This enables new product and service offerings as well as entire revenue streams that were not feasible or possible through the company's traditional business activities.

In the "digitally transforming" concept diagram above, the dependency between the two activities, "traditional" and "digital," has been removed. Obviously, the traditional activity must still be supported by software—perhaps even in an enhanced way. However, the digital activity may not depend at all on the traditional—at least not in the same way it did before. Digital now has its own engagement models, customer base, and revenue streams. It is no longer "fed" by the traditional business; instead it stands on its own.

As a concrete example, look at Amazon as a retailer compared to the now-typical web- and mobile-enabled department store. The digital-enabled department store is using digital as a channel, but it is not acting as a "digital native". Like the department store, Amazon also generates revenue by selling

its own products, and it uses software to do so very efficiently, through a variety of channels. However, as a digital native, Amazon also captures new revenue streams. It does this in many ways, including "renting" digital space to other sellers and capturing a share of their revenue, and by selling subscriptions to buyers, who receive preferred treatment (expedited shipping, for example). Amazon even drives revenue from renting out its peak computer capacity.

These and other additional sources of revenue are purely digital. While each revenue stream has some analogs in the physical world (triple-net leases and in-store boutiques, membership at Costco, "co-lo" hosting), they have little or nothing to do with a traditional retail business whose revenue comes from selling physical products to consumers. That one business entity can capture these novel sources of revenue at scale and on top of their traditional business activities is something only a digitally transformed / digitally native company can do.

The novel revenue streams and business models enabled by digital are such that a digitally transformed company's "digital" business is, or could be, entirely independent of the "traditional" business. One key indicator we use to identify a successful transformation is when a business starts to at least consider the idea of launching their digital business as a separate entity. Whether or not such a spin-off actually happens is a business decision that depends on many factors. The important point from a transformation standpoint, though, is that it could be done. The digital business could stand on its own as a viable and prosperous business entity—that's what success in digital transformation looks like.

Given the obvious benefits to a business that come from digital transformation, why is it so hard?

A transforming business has two components: those engaged in its "legacy" business activities (brick-and-mortar retail, commercial banking, etc.), and the nascent digital business. In any organization, power and budget tend to flow to those generating or controlling the largest sources of revenue or other resources. Because the early stages of digital require investment and generate relatively little revenue compared to the overall business, it tends to have little organizational power. Even as digital begins to grow, it takes a while for the power dynamic to

catch up. The net effect is that instead of reaping the rewards of its own growth, revenue from digital tends to be syphoned off to sustain the legacy, rather than used to fuel growth of the digital upstart.

To overcome this dynamic, a leader must believe, or a consensus must exist within the organization, that digital is the future, along with the will and authority to drive transformation forward, even against opposition. It's the rarity of such a leader or consensus that makes digital transformation so challenging to many existing organizations.

Another reason many companies have not yet transformed is that it's hard to recognize at the time whether a technology is transformational or incremental. Technology changes constantly, and overwhelmingly each of these changes is incremental. However, the digital tsunami has clearly reached the shore. Whether this is due to one specific innovation or the cumulative effect of many is now incidental. There is no question that as we write this, the world around us has changed profoundly in terms of its most important characteristics: Our means of interacting with business, information, the physical world, and each other. Responding to such a transformational change with an incremental response guarantees failure, immediately or mid- to-long-term.

When confronted with any truly transformational technology, the most successful response is not simply to use it to do the same thing you do today better, faster, or cheaper. The most successful response is to ask what new activities the transformative technology enables, and then grasp those opportunities. Positioning yourself technically to seize the genuinely new possibilities presented by digital is the functional definition of "Digital Transformation" we will use in this paper.

Time and Cost Profiles

In our previous sections, we discussed the three major transformation approaches available to you as an engineering leader. We also discussed what the end state looks like and what it means to "digitally transform." Our primary goal in our next several sections is to help you connect those dots and choose the transformation approach that will lead you from where you are today to your desired, digitally transformed end state.

Let's start by taking a qualitative look at the time and cost side of the three major transformation options we discussed in the previous section, Greenfield, Side-by-Side, and Gradual Evolution. In subsequent sections, we will analyze and illustrate each of these approaches in detail.

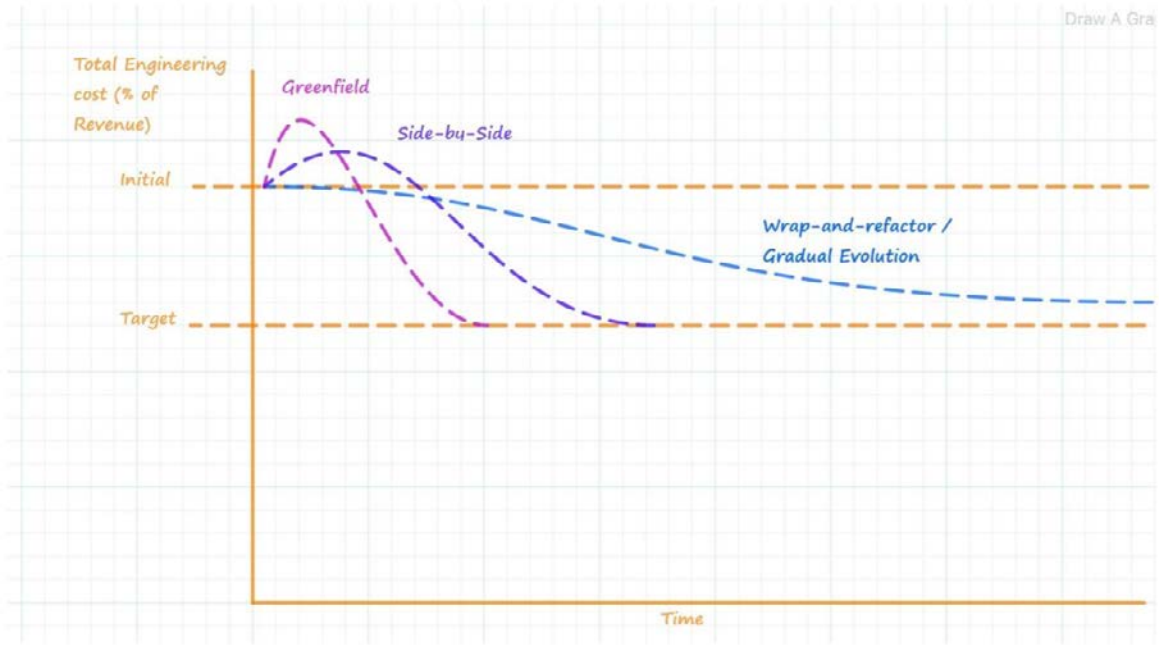

Figure 3: Cost curve of the three primary technical transformation approaches

For an established company to stay competitive, your current system(s) needs to be actively operated, maintained, and enhanced while you are replacing it. If your goal is to cut engineering costs as a share of total revenue, there is no way around the fact that, all things being equal, you will spend more money on

the Greenfield or the Side-by-Side approach than you are spending today—at least initially. This is because you need to keep doing what you are doing, and you also need to staff and run a second engineering effort in parallel. You will obviously try to minimize this "bump" by reducing your investment in maintenance of your legacy. In general—and for multiple reasons—you will see an increase in overall spending before the cost curve bends down.

The good news with either the Greenfield or the Side-by-Side approach is that your time to bend the cost curve down is relatively short. Every situation is different and depends on many factors, including financial, cultural, technical, business conditions, and many others; therefore, there is no absolute time frame for transformation. The one universal you can count on is that it won't be as fast or as cheap as you wish it was.

It's always risky to do this, because it's like asking "how long is a rope"? Still, to give some feel for the numbers, here are a few that are representative of what we see in real life: For a sub-$1B (USD) revenue company fully committed to transformation—that is, where a single owner has full control of the product team and transformation is their top, uninterrupted priority—you might reasonably hope to begin retiring elements of your legacy system in about 9 months if you take a Greenfield approach or, with a Side-by-Side approach, have your NextGen system at feature parity with your legacy in about 18 months—maybe.

For a multi-billion-dollar (USD) company with a complex structure, massive code base and equally massive quality debt, the time for a fully-committed initiative to really start to bend the cost curve down could be around 3 years for a Greenfield initiative, or about 5 years for a Side-by-Side approach. Sorry to be the bearer of bad news. While your situation might be, and probably is, different than we've ever seen before, I would not charge into a transformation initiative with the belief that your actual results will be dramatically better, unless you have a history of similar successful initiatives at that *same company* to back you up.

We will describe the reasons a Side-by-Side approach takes longer than Greenfield—but adds less incremental cost per unit time—later on.

Note that these "strawman" figures above assume your current system is really a mess, which is typical when a transformation initiative is launched. Companies that religiously refactor their code, pro-actively integrate new acquisitions, keep current with the latest technologies, have full end-to-end test and deployment automation and monitoring already in place, consistently focus on "quality first," have meticulously updated system descriptions or TDD tests to orient new developers, and have been fearless and unafraid in modifying their system architectures as improvements are needed can pivot radically almost instantly.

This is the goal we all aspire to when we launch a transformation initiative,and it can be achieved. At the beginning of such an initiative, though, it will take an injection of time, money, and sheer will to get to this place. This is because in many situations you must undo the effects of expedient quick-fix changes, shortcuts, and deferred maintenance and modernization that have often accumulated over a period of many years. You must also overcome the organizational and other factors that led to these issues with your current system in the first place.

A Gradual Evolution approach is the third fundamental technical approach, and the only one that has the potential to begin reducing costs almost immediately. However, the gradual evolution approach is exactly what the name implies: Gradual. One multi-billion-dollar (USD) company who recently became a client began a "gradual evolution" to SOA in the mid-2000's that is *still* not completed *over a decade later.*

This situation presents an interesting conundrum for that company: Do they complete their transformation by migrating to a technology (SOA) which is itself now obsolete? Or do they start over again with another refactoring project, layering on another technology stack and leaving their already-converted systems in a partially migrated state forever? Far from pointing the finger at this organization, there are very good reasons why Gradual Evolution projects are often never finished, which we will also explore later.

Before we close, a word about the case studies we will present in this paper. We believe using real-life scenarios to illustrate the various transformation approaches is the only way you can truly judge their applicability to your own situation. Out of respect for our client's confidentiality, these case studies will be heavily obfuscated, to the point where (we hope) it should be hard to even identify what industry the client is in, let alone the specific client.

This degree of obfuscation is important because many of these companies are leaders in their respective industries, so knowing the segment means you could potentially guess the company. In fact, our goal is that even a client who is the subject of a case study should not be able to tell for sure whether or not we are talking about them. This is easier than it might seem because situations tend to recur! We do not believe this obfuscation detracts from the essential points—especially as none of the situations we describe are "ideal" since they describe real-life companies and real-life situations. We hope this reality-basis helps you both relate to them and apply them to your particular situation.

The Greenfield Approach

Greenfield Approach

In the introduction, we defined three key approaches to digital transformation, which we call Greenfield, Side-by-Side, and Gradual Evolution. We also outlined their relative cost profiles. In this section, we focus on the pros, cons and "economic" profile of the first of these approaches, the Greenfield approach.

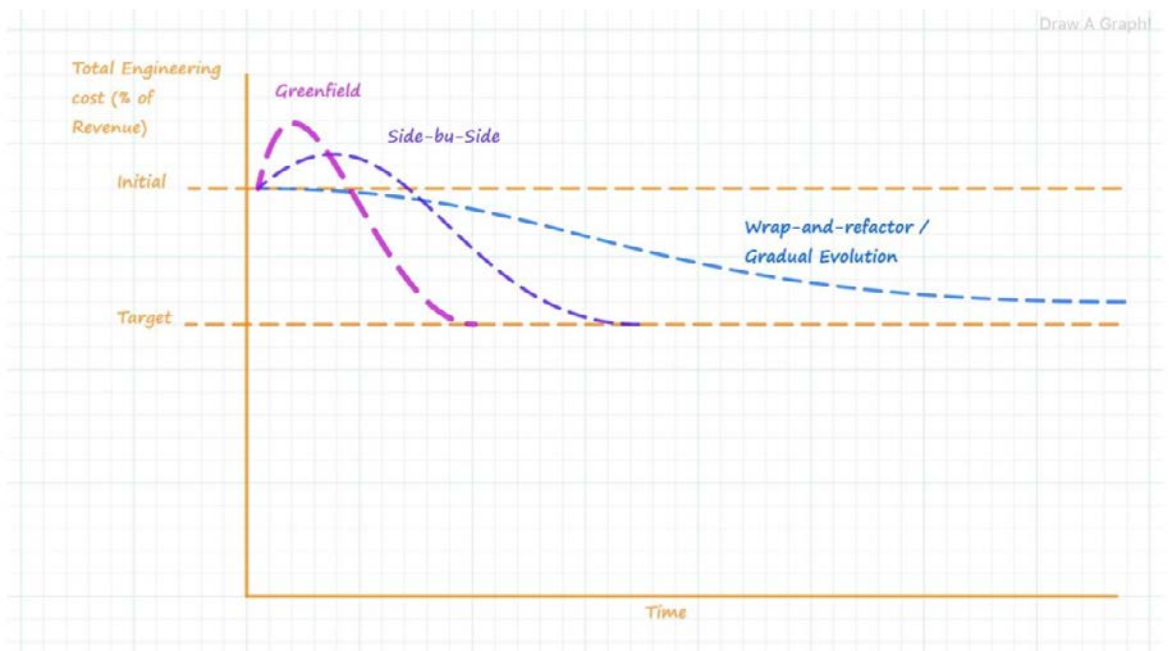

Figure 4: Greenfield cost curve

The Greenfield approach tends to be the fastest and, perhaps surprisingly, the overall cheapest path to digital transformation and system modernization. It also has the potential to create the biggest upside revenue growth the soonest, because you are unencumbered by legacy code and can move as aggressively as a start-up in delivering a first-rate system, but with the benefits of your existing business know-how and customer base.

The argument that starting from scratch can be faster and cheaper than starting with what you've already got may be counterintuitive. If you are a large company, you've already invested a lot—in fact a literal fortune—in your legacy system. The idea that "throwing it all away and starting over" (so it seems) could get you to

the end state faster and cheaper seems unlikely at best. However, the greatest effort in many engineering transformation initiatives goes into largely futile wrestling with idiosyncrasies of the established code base. That's why startups can seemingly move so fast.

In many complex and established systems, the code may have grown up over a period of years or even decades. It may partially incorporate now-obsolete technologies and have been developed by generations of engineering staff who are no longer on board. Complex systems often have expedient integrations with third-party and acquired systems that are fragile and add to the overall complexity.

The opportunity to "start fresh" using modern technologies often comes as a breath of fresh air to an engineering team. Since the late 2000's in particular, there has been a revolution in technologies that can radically reduce the effort required to produce a new system. These advances include many buzzwords you are familiar with: Cloud, NoSQL, containers, Microservices, and event-driven architectures, to name a few. If your system was implemented more than a few years ago, it is a certainty that much of its complexity was introduced to work around the limitations of then-available technologies.

Specifically, this means your team is now spending time and money maintaining code which would benefit your system more by being thrown away and replaced! Attempting to carry your entire code base forward into the next generation is simply a waste. The Greenfield approach leaves this "technology debt" behind and enables you to be as nimble as a startup.

The Greenfield approach is also the only one that has no impact on your current code base, as it is an entirely separate initiative. At most, current code can be an "organ donor" to the new system, with existing components being harvested where they genuinely help speed the effort. However, such harvesting has little or no impact on either the production code of those existing systems, or on the teams enhancing and maintaining them. At worst, if managed well, harvesting is a minor distraction.

This lack of any direct impact on the current system(s) means those systems are free to evolve at full speed to meet market and customer demands with current skill sets. This is no small advantage, since most businesses need to keep

operating and enhancing their existing systems vigorously to remain competitive with other players, and they must achieve this for the entire time the next-generation system is under development.

On the downside, this parallel evolution of systems means the new enhancements to existing code bases must either be ported or re-implemented in the NextGen system in order to achieve full feature parity, which makes the definition of "feature parity" a moving target. However, making the new system as featureful as the evolving old system is a relatively minor effort compared to the other alternatives. In particular, trying to enhance and simultaneously refactor the same deployed code base ("change the wings on the plane while it's flying"), which other approaches can require, is a daunting task that we will discuss later on.

The alternative to continuing to enhance your current systems would be to NOT keep your current system competitive with the other players in your market during the time the new system is under development. While this may be possible in some niche areas, or in some captive markets, few industries are so mature or niches so static that standing still feature-wise can be done for long without negative impact on the enterprise.

For most enterprises taking a Greenfield approach, there is no way around supporting two (or more) vigorous, full-speed, fully-funded development efforts in parallel—one pointed at the future, the other actively maintaining and enhancing the past. This parallelism is the primary source of the "cost bump" in the Greenfield approach, with savings and business benefits only coming once the old starts to be transitioned to the new.

The Greenfield approach is the only one that keeps your currently deployed systems and enhancement activities totally undisturbed with no direct impact on your current business or engineering teams. This is also a negative factor, however, because it (a) drives incremental costs during the term of these parallel efforts, and (b) has minimal direct transformational impact on your current engineering teams, since they may not be involved in a Greenfield project at all. While such projects can be a "lighthouse" pointing the way to the future, they are not necessarily a training ground for your current teams. This training can be done, but it needs to be a conscious effort.

Greenfield Approach
Case Study

In the previous section, we discussed the Greenfield approach to digital transformation. In this section, we will present a case study where a client applied this approach. As we noted in the introduction, we have heavily obfuscated this case study to protect the client's identity, market segment, and confidentiality.

One client who adopted a Greenfield approach to digital transformation is a multi-billion-dollar (USD) company who, like many large companies, had grown both organically and by acquisition over a long period of time. Historically, the company's primary business was in the sale of physical goods and related services, though revenue from these traditional areas was beginning to decline as competition from digital natives increased.

As a large company, the client was an early pioneer of digital in their space though, digital was seen more as a supporting role to their physical business than a primary business activity in itself. As the competition was increasingly "digital only," this perspective was beginning to shift.

The net effect of the company's organic and inorganic growth on their existing digital platform(s) was to increase them in number. Over time they built or acquired literally dozens of revenue-generating customer-facing platforms that—from a functionality perspective—all essentially served the same purpose. Some of these addressed the needs of a particular market sub-segment, while others had a different interaction paradigm; still others represented earlier failed or partially successful attempts at platform consolidation.

All the active systems had end users, and they all required on-going maintenance and support as well as hardware, operations, and people resources to keep them up-and-running. The net result was this client was spending hundreds of millions of dollars per year simply to keep the digital lights on and to try to stay competitive with emerging players and "lateral entrants" into their space.

While an active acquisition and investment agenda allowed the client to leverage some of these new innovations and mitigate competitive pressures by actively enhancing some of their platforms, the sheer inertia and cost of maintaining the great bulk of their legacy systems limited how fast they could move. Despite remaining a major player in their market, it was becoming clear to senior management that their historical leadership position was actively eroding. This was also reflected in the stock price which, at one point, dropped to less than 50% of its peak value the year before.

Bottom line, this company was now open to a Greenfield approach primarily because they had tried the other approaches and found them wanting. They had already tried incremental evolution and "side by side" enhancements, and their CEO, CIO, and CTO were honest enough to admit those earlier efforts had failed. The clients' resources, knowledge, and reputation were such that they could win—or acquire the winner—in any specific niche where they chose to compete.

However, establishing a position of sustainable dominance for their industry as a whole would clearly require the creation of something that simply didn't exist today. That is what a Greenfield approach had the promise to do and is why they chose it.

Perhaps even more important to this client was the fact that their market leadership position was eroding, to the extent that their ability to deliver on their mission as a company was beginning to be impaired. Other more nimble players and large "digital native" technology companies making lateral moves into their market were beginning to claim leadership roles in various sub-segments. These players were able to offer different channels as well as different pricing and business models which end-users found attractive.

The mandate from the client when architecting, designing, and implementing their next-generation platform was specifically not to reuse any existing code or architectural patterns. The client's thinking was that since past solutions had failed—completely or substantially—use of existing code or design patterns would also compromise the next-generation system.

While this point of view has merit, there were some subsystems design patterns worth salvaging from the best of the client's (many) current-generation platforms. Ultimately, the paradigm that emerged was to reuse legacy code or designs if and only if it improved the next-generation system, and then only after independent review with explicit approval by client execs. Reuse for the sake of reuse was strictly forbidden in this project, which made it a very pure Greenfield approach.

While total avoidance of existing code and design patterns may seem extreme, keep in mind the client's whole purpose was to introduce a "game changer" for their company and their industry. Approaches that had been tried and found wanting were automatically suspect. The client's goal was to deliver on their vision for a next-generation system that would place them firmly in a position of industry leadership. Last-generation thinking was decidedly not welcome; the entire focus was on the future.

Also, perhaps not so obviously, the decision not to reuse existing code (later relaxed to using it only in closely-monitored circumstances) limited "political" reuse decisions. Within a large organization, the goal of maintaining the size and influence of one's team often becomes an end in itself. Facing potential obsolesce by a next-generation system, one way to maintain relevance is to have your current system incorporated into the next-generation system to a greater or lesser extent.

While in some cases this is technically warranted, in others it is purely political and—like a sea-creature encrusted with barnacles and parasites—can end up compromising the next-generation system severely. Making reuse decisions on a technical rather than political basis is essential to the success of the next-generation system. The client's mandate not to reuse existing code had a secondary effect of limiting "political reuse" to the largest extent possible.

Working from an empty whiteboard / blank sheet of paper allows product managers, architects, and designers to do their best work. It is both liberating to the team and highly beneficial to the organization, since current best-of-breed technologies and approaches can be brought to bear. In a large organization, politics, team structure, prior relationships, and other constraints often get in the way of choosing the best technical approach and architecting the best solution.

In this case, the team found it a breath of fresh air to not only be allowed to think outside the box, but to actually be required to do so. Even so, and even with the backing of the most senior executives, addressing demands for "political reuse" remained one of the key challenges that needed to be dealt with throughout the initiative.

The client took the same Greenfield approach to the implementation team—it would be completely separate from the existing engineering organization and staffed almost exclusively with new hires. Some technical oversight and input came from existing staff, but their bosses and the overwhelming majority of the technical leadership were brought in from outside, primarily from engineering leadership roles at Silicon Valley product companies.

Product management resources were drawn from the existing organization, because they had the domain expertise. The current-generation product managers were supplemented by business analysts with startup and "1.0" experience gained outside the company. Ultimately this helped bring about a startup mentality, though it took time to achieve.

For the most part, the new team was able to work independently of the existing organization structure, while still leveraging the client's deep domain knowledge and expert resources. This did not work perfectly, but since the client needed to continue operations and maintenance of the existing systems while the new one was under development, most current teams were fully occupied with roadmap items and keeping the lights on.

As it became clear the new platform would be successful, however, the distraction value went up as some teams made plays to remain relevant in the new world by inserting their code (or teams) into the new system. This led to some political challenges for reuse and ownership that needed to be addressed.

The architecture for the next-generation system had a variety of proprietary aspects specific to the client. Some of these will eventually be disclosed through the workings of the patent process, but for the sake of confidentiality of the client's IP, we will not describe them here. However, at a 50,000-foot level, the

bare bones of this next-generation architecture has some common features with the following enterprise reference architecture:

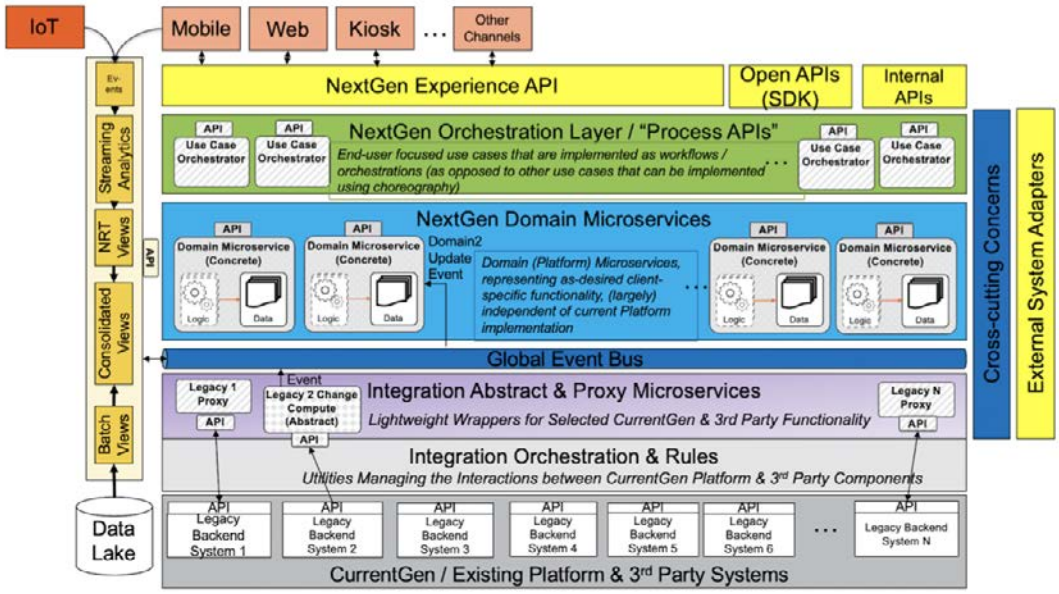

Figure 5: Enterprise reference architecture

Because the Greenfield approach allows a fresh start, the next-generation architecture could fully leverage the, then proven but (at that time) still emerging, cloud-native containerized microservices paradigm. This approach was ideally suited to the client's goals of massive scale, high-reliability, global reach, and system portability. The architecture was built up in layers, with the "lowest" level consisting of microservice "wrappers" around legacy and 3rd party external services.

On top of those wrappers were a set of "domain" microservices that implemented the client-specific business and functional abstractions needed to create the solution. The domain microservices communicated with each other either through an event bus which supported the "choreography" of services, or through API calls made by domain-level orchestrators. At the top level of the platform, a set of orchestration components were available to implement workflows and business processes (which were not practical to choreograph) using orchestrated calls to the APIs of the domain and wrapper microservices.

In addition, as is increasingly common, the client wanted to capture and react to asynchronous events initiating from mobile and Web clients, as well as (potentially) from devices and information sources external to the system. To process and respond to those events, we implemented a Marz "Lambda" architecture to do near real-time, context-aware event ingestion, processing, and response. Processed events could also be provided to other microservices via the shared event bus, and analyzed data could be accessed via an API.

Given the layer of "wrapper" microservices, it might be surprising that even a Greenfield system relies on legacy and 3rd party systems. This is because no modern software system truly exists in isolation, even a Greenfield platform. The dependency boils down to the somewhat philosophical issue of "what is the platform?" or, more specifically, "what are the boundaries of a given Greenfield platform?"

In general, we believe companies undertaking a Greenfield initiative should define their platform boundaries to include the elements of a total system that address the unique or value-creating aspects of their business or those that cannot otherwise be readily bought or acquired. The microservices which implement the most differentiated use cases are the "core" systems, represented in the reference architecture above by the "domain" services. The "supporting" systems are generally not significantly altered; instead they are "wrapped" for easy access by the core services and to enable possible retirement or exchange of those external systems in the future.

Whether a system is "core" or "supporting"depends on the business you are in and the business value you are trying to create. For the client in this case study, for example, there was—characteristically for them—no single user identity management system. Rather, there were several similar systems that had been developed independently over time, each used by a partially overlapping set of legacy platforms. Not incidentally, each identity management system managed its own set of users in a unique way.

This duplication definitely had an impact on the client's core business, because an end user required multiple identities (and credentials) to use more than one of the client's platforms. This was not only frustrating for end users, but it also impeded our client in their attempts to monitor customer usage across

platforms. It also multiplied the client's administrative and support costs because of the redundancy.

While we ultimately concluded that converging from the current state to a "single identity" was an important initiative, but outside the scope of the Greenfield platform, it's clearly a judgement call. What we ended up doing was using identity token to uniquely identify a user within the next generation platform, but external the association of that identity token with a particular set of credentials. In other words, it was up to each existing "supporting" identity management system to map a given set of credentials (username and password, for example) to the correct, unique identity token denoting a specific individual in the next-gen platform.

This enabled multiple logins (e.g. multiple usernames and passwords) to map to a single identity in the new platform. While this did not solve the legacy multiple-credentials problem for the client's current systems, it did prevent proliferating the problem of having multiple identities for the same person within the new platform. Sometimes pragmatism dictates that the best you can do right now is to avoid making the current problem worse. Solving a problem that was decades in the making (as in this case) is not always something you can or should take on within the scope of the core Greenfield initiative.

While all the transformation approaches have the potential to change the client's business model, and change the game for the industry they are in, the Greenfield approach has the biggest potential to be transformative quickly. However, as we mentioned previously, a major challenge for the Greenfield approach is that it is pure investment until the product is deployed. There is no direct business benefit from the work until that point—just cost.

This client took the rather risky-seeming decision to invest in a Greenfield solution, but they did so in a phased way and had several factors operating in their favor:

- The total investment—though large in dollar terms—was modest compared with the client's annual engineering spend. While the cost was significant enough for visibility at the board level, it was not so significant that a failure of this project would by itself cause the company to fail. In other words, while it was definitely a "bet your job" proposition (as they frankly confessed) for

the C-level executives supporting it, it was not an "expose the company to financial ruin if it fails" level of investment.

- Had the project failed, the digital future of the client would have continued to remain in question, as it had been before the project started. The client's future prospects (and valuation) would have therefore almost certainly continued on the downward path they were already on. However, the size, reputation, and momentum of the company in its market were such that they clearly could have sustained a gradual decline supported by their current technology for some years—maybe even for enough time to try again.

  Even if they never successfully transformed, many employees could have ridden the company down until it was acquired, went under, or found success as a smaller entity. So, while important, and while a potential career-limiter or -ender for some, the rank-and-file were not panicked by concerns about the project's success or failure. This sense of security is important when taking a transformation risk so undue pressure is not put on the "next-gen" team, and they can do their best work.

- The project was originally championed by the client's CTO, whose first step was to create a next-generation architecture and get the CIO / COO on board. Together they championed the development of a working proof of concept ("POC") based on the new architecture. The success of the POC enabled these executives in turn to "sell" the approach to the CEO, who then brought it to the board with his endorsement. Other C-level executives were persuaded to be at least neutral to positive and broadly willing to try the approach.

- At the board level, the company recruited new members with deep technology backgrounds because it was clear to all concerned that the company's future—if it was to have a bright one—would be based on its success in the digital realm. These members were also generally supportive of a more radical approach that would catapult the client to a position of true leadership in the digital realm, as they had been for many years in the physical. Not incidentally to all concerned, this company also had a sense of mission it wanted to fulfill in the new digital world.

- The management team has proved to be stable throughout development.

> Changes in leadership can end or redirect an initiative regardless of its current success or inherent merits. While there's no way this management consistency could have been guaranteed at the beginning of the project, it did happen and was an important success factor. In fact, one of the executive sponsors received a major promotion based in large part on the success of the transformation work.

While desirable in any transformation scenario, a broad base of support is essential for a Greenfield approach because in the development stage it is pure investment. The only signs of progress are engineering status reports and internal technology demonstrations. No revenue is generated until the system is deployed in production, which may require business model changes, sales training and new incentive structures, data and user migration, end-user education, internal and external marketing, and many other non-technical and operational hurdles to overcome.

How has it worked out business-wise for the client? After they announced a digital strategy centered around their next-generation platform, the client's shares started trading more than 60% higher than when they began the initiative. While the share price is still below its historical high at this writing, they appear to be on an upward trajectory.

While this improving financial outlook is encouraging, the truth is it will literally take years to decommission and replace all of the client's dozens of legacy platforms. Combined, these systems would comprise hundreds of millions of lines of code and are supported today by literally thousands of engineers and technical staff. While the client now has a clear technical path to the future, the key challenge becomes: how fast can they shed the past? The past tends to fight back in business, and many whose roles will change will undoubtedly refuse to embrace the future or will actively oppose it.

Consider Apple's transition from their legacy Mac operating system (Mac OS 9) to the then-revolutionary OS X in 2001. It took the steely resolve of Steve Jobs to take the company through that transition and to "shed the past" of the last-generation system. Once that core next-generation technology was adopted, the new OS X platform became the enabler of Apple's future business success

by serving as the technical basis for the iPhone and iPad (iOS), wearables ("watchOS"), TV and home automation, etc. These pivots and new products could not have been achieved with Apple's legacy system as their technical foundation. The old system had to go for Apple to end up where it did—with a trillion dollars of market cap in 2018 and the share price more than 125x its value when OS X was introduced.

Similarly, using their new platform, this client can be as nimble as any startup. They are well positioned technically to gain the digital market dominance they historically enjoyed for their physical offerings. Architecturally, they are also well positioned to pivot to grasp future opportunities that are not even foreseeable today—as Apple did so successfully with their new and far more flexible operating system "OS X".

However, we all know there is no certainty in business. The client must continue to find the will and the courage to shed the past and complete its transition to a fully digital company. This takes more than technology. However, the Greenfield approach they chose to adopt has given them the technical underpinnings that can lead them to future success.

As we will caution for the opposite outcome in the "Gradual Evolution" case study, do not judge the value of the Greenfield approach by its apparent success in this situation. Success is the result of a client choosing the right approach for their particular needs and situation. The Greenfield approach comes with its own challenges and limitations. We will discuss some of those in the next section.

Greenfield Approach
Pros and Cons

In the previous section, we presented a case study describing one client's experience using a Greenfield approach to digital transformation. While it does indeed have the potential to position a company for great success, the Greenfield approach is no panacea. Some of its downsides include:

- **Low inherent transformational impact on your existing organization.** Given the need for new skill sets, a Greenfield project is often developed by a separate team in parallel with your existing operations. This team can consist of new hires, retrained existing staff, or both.
  Having parallel teams is both an upside and a downside in terms of transformation. We have discussed some of the positives in previous sections. On the negative side, your "legacy" team will not, by default, be dragged into the future by the Greenfield project. The team actively engaged in the Greenfield project does indeed need to acquire—or already possess—new skills. However, for the remainder of your legacy team, neither skill sets, process, nor behavior necessarily need to change for the Greenfield project to succeed.

  While the lack of inherent transformational impact can be mitigated, it will leave you with a skill set gap between "old" and "new" if it is not consciously addressed. If your current system is managed by a 3rd party, for example, the transformative nature of the development process itself may not be a key consideration for you. Alternately, the downside might simply be outweighed by the need to get to market with a new system quickly.

- **Increased short-term costs.** To implement a Greenfield approach, you will need two teams. One team "keeps you in business" by maintaining and enhancing your legacy system. This needs to be done to keep your current revenue flowing, as well as to meet customer and roadmap commitments while the "NextGen" system is under construction. In parallel, you will need a second team to implement the Next Generation system.

While we've made the point that the Greenfield approach has the lowest cost overall, that sense of the word *overall* is the value of the (time) integral over the time-cost curve. The integral (area under the time-cost curve) is the smallest because the curve bends down sharply once you begin to "shed the past." However, in the initial stages, your costs will go up while you "build the future." Investing now for savings later is exactly in line with the culture of some companies; but for others the notion is regarded as ludicrous, dangerous, or worse. Again, the art here is choosing the approach that best fits your internal and external situation. Even if Greenfield is otherwise right for you, if you can't "sell" it internally, you may need to choose another path.

Another path to "selling" Greenfield is to try to reduce the size of the initial cost increase. While you are creating your NextGen system you may be able to reduce the near-term cost bump in several ways, depending on your business situation; this can sometimes make it more palatable when it's the right thing to do. These cost reduction possibilities include: putting the Legacy system(s) on "life support" (critical bug fixes / urgent features only); shifting NextGen and/or legacy work to lower-cost geographies; taking staff off legacy and re-deploying them to NextGen or another project; or even by canceling your legacy roadmap altogether.

Despite these or other mitigation efforts, though, there is still generally a period of time when you will have increased costs with a Greenfield approach; it's simply the nature of the beast. We discuss a number of these costs in more detail below, but as an overview they often include:

Retraining your existing team in new technologies and new ways of working (for example, Agile / SAFe)

Recruiting, hiring, and ramping-up for new skill sets—whether co-located with your current team(s) or in new geography(s), together with associated overhead in facilities, equipment, on-boarding, etc.

Paying for the new skill sets, and dealing with resulting inequities with your current team. Within a given labor pool for a given level of experience, technical staff skilled in the latest technologies tend to cost more to attract than people with conventional skill sets. Also, for a given

labor pool, new hires tend to cost more than comparable current employees, even for the same skill set. Finally, hiring new people at higher salaries than your current staff generally forces you to either give raises to existing employees you most want to keep or provide other (monetary, equity, career advancement, etc.) incentives. All of these factors tend to cost money and increase engineering costs

Purchasing, evaluation, and contract negotiation for new technologies and new tools, together with infrastructure setup and configuration, including familiarization POCs and other experiments

Inefficiencies and burnout as existing key personnel jump back and forth between the new system while still being pulled in to solve customer and other urgent issues on the legacy system(s)
Management and "key staff" overhead of planning work on two different systems at the same time, with very different technology and speed profiles

Incurring the operations costs for your legacy system(s) until they can be fully phased out

Migration costs from Legacy to NextGen—these include data migration, customer migration, support training, and many other factors

Product Management and BA time and staff to reverse-engineer your legacy system

Product Management, Architecture, Design, and management time to visualize, design, and plan implementation of the system you actually want.

In addition to these expenses, your organization may have other imperatives, including compliance costs, contract reviews of new vendors, partners, open source and cloud providers, and many others. While every system and company is different, it would not be unusual for a Greenfield transformation strategy on a complex system in a small to medium sized (several $100M USD) company to increase overall engineering costs for at least a year, and

for at least three years for a larger (>$1B USD) company.

- Requirements and reverse engineering time and cost. The essence of the Greenfield approach is that, just like a startup, you are implementing a new system from a blank sheet of paper. While this presents a tremendous opportunity to get the system you really want, it also requires you to *describe* and *design* the system you really want. In other words, you must create designs, requirements, and a next-generation architecture for your new system. This takes time, money, and often new talent.

  As an established company, you are in a much better position than most startups in your field because you already have working systems and deep domain knowledge. However, very few companies have good up-to-date requirements and designs characterizing their current systems. Even when these do exist, they tend to be descriptions of how your systems *currently* work, rather than descriptions of how a next-generation system *should* work. Taking full advantage of the possibilities that new technologies offer requires a complete rethinking of your system architecture—and you may or may not have that talent already on-board. We will discuss this more below. Here, we note that you will also need to create or strengthen a software product management and experience design function within your company.

  Some of your existing BA's, Testers, and even current-generation developers will have the requisite domain knowledge and are likely to be good candidates for roles within a strengthened product management function. However, the mindset of a current-generation domain expert and that of a forward-looking software product manager or designer are quite different. It is a challenge for many domain experts to make the leap from understanding how the system *actually* works today to describing how it *should work* in the future. Some current system experts prove great at describing the system they *really* want; others are puzzled by the whole concept.

  Many domain experts don't think in terms of capturing business opportunities; they tend to be technicians in their own way. Often, to create an effective organization, you will need to hire experienced software product managers and designers from outside your company, or even from outside your domain, and supplement the new people with current staff having appropriate domain skills.

While a Greenfield approach gives you the *opportunity* to create a system that delivers what your company really needs, the time and effort it takes to "reverse engineer", re-imagine, and design your next-generation system has a cost. Even for companies with an established product management function, introducing a modern software-focused, design-led, customer and business-centric software product management mindset is a big cultural change.

- New skill sets for engineers and architects. To take advantage of the most recent technologies in the NextGen system, it is very likely that you will need to develop or acquire new technical skill sets. Good engineers and architects will rapidly come up to speed on the nuts-and-bolts of many of the new technologies. However, to fully exploit their potential, many modern technologies require new programming skills and a different way of thinking. For example, there is a big difference between a cloud-native, distributed systems architecture, and a conventional RDBMS-centric architecture that happens to be hosted on the cloud. The latter may be "buzzword compliant" with the Cloud, but it will lack the business benefits of elastic scalability and robustness provided by a real cloud-native system.

  Initial inexperience with a new paradigm is not in any way a lifetime sentence for a good engineer. Smart and aggressive engineers and architects with open minds can start to become proficient in these new paradigms in a matter of months. The new technologies are, after all, improvements—and doing things better is what gets good engineers and architects excited.

  However, it takes time, and not all your engineers or architects will make a full transition to the new paradigm. Some will be stuck in the old ways of doing things and never become effective on your "NextGen" technology stack.

  Bottom line, you will probably need to bring in new staff to work on your new Greenfield system, and you will definitely need to bring in people to help them learn. You will also need to take some of your best people off your legacy system and retrain them, losing their full productivity for some time and forcing you to either back-fill the legacy team or slow down the pace of legacy development. You will also need to budget time for training and decreased productivity until the team becomes fully proficient in the new ways of working.

- No business benefit until it's done. Perhaps the biggest drawback of the Greenfield approach is that there is no direct business benefit until the NextGen system is actually deployed in production. Until then, it's a pure cost / investment.

  Using Agile delivery will allow you to demo and pilot intermediate releases and deliverables, and these can generate good will and excitement internally and externally. Sometimes you can even stage early "partially complete" releases that provide business benefit and show momentum to customers and prospects. However, until the NextGen system is at or near feature parity with the Legacy system(s), it cannot replace it. This, plus the incremental up-front costs, can make a Greenfield approach politically non-viable at some organizations, however desirable it may otherwise seem.

While your absolute money costs may be higher during development, there is often no better way than a Greenfield approach to get a genuinely new system out the door quickly. If speed helps you capture market share or lessens the erosion of your current market share, the "bubble" cost of Greenfield may be more than offset by increased or protected revenue.

This is one reason for the steep decline of the Greenfield cost curve; another is that in many situations when modern technologies are used properly, engineering and total operations costs are genuinely lower at scale than for traditional systems. The major factor, though, is that ultimately having a Greenfield system in place allows you to migrate customers off of, and ultimately retire, your legacy systems, shedding their associated costs.

While creating an effective "software product management" function within your company is an overhead, it is also an opportunity. We live in an age where every company needs to act like a software company, and software companies have software product managers. Unlike an IT "project", a revenue-generating software "product" is never done. It will be continually enhanced and improved as long as you continue to engage in the business activities the software embodies. If your company is going to start acting like a software company, then a Greenfield project is a good opportunity to begin.

In supporting product development, we've seen that product management skills from other domains can transfer to a new domain surprisingly well. For

example, we've seen software product executives from unrelated industries like telecommunications, media, and video gaming become highly successful in retail, education, consumer packaged goods, and other "verticals". In many cases, these people became highly impactful in a matter of months. While every domain is unique, they don't tend to be so unique that a good product manager or technology executive in one domain can't rapidly become a good one in another—especially if they have your existing domain experts to lean on.

Pros	Cons
Generally, the fastest and overall ("integral under the curve") cheapest path to a modern, technologically advanced system	Costs are incurred before business value is generated, which may be a political non-starter. Perceived as a "high risk" in some business cultures.
Knowledge gained from past systems is leveraged, while the new system implementation itself remains uncompromised by legacy issues and "quality debt"	Requires high degree of domain knowledge, plus investment in product & program management and design to envision and define the new system
Does not add complexity to the current architecture since the NextGen system is developed independently	May need to write-down technology assets being replaced
Since they are separate initiatives, you can maintain a full-speed development roadmap for your currently deployed systems while simultaneously working on the NextGen system. This can keep you competitive while NextGen is under development.	Likely requires new architecture and engineering skillsets since technology will differ from legacy
Maximum opportunity to "change the game" by introducing new business models, products and services.	Low "inherent" transformational impact on people not directly involved in the Greenfield project

A "Green Field" approach is best used when the key goal is technology or industry leadership or rapid parity with a superior technology. The company and stakeholders must be willing to tolerate risk, up-front investment, and some staff "left behind"to get to a new place quickly.

The Side-by-Side Approach

Side-by-Side Approach

In the introduction, we defined the three key approaches to digital transformation, which we call Greenfield, Side-by-Side, and Gradual Evolution. We also outlined their relative cost profiles. In the previous section, we focused on the Greenfield approach. We now turn our focus to the pros, cons, and economic profile of the Side-by-Side technique of digital transformation.

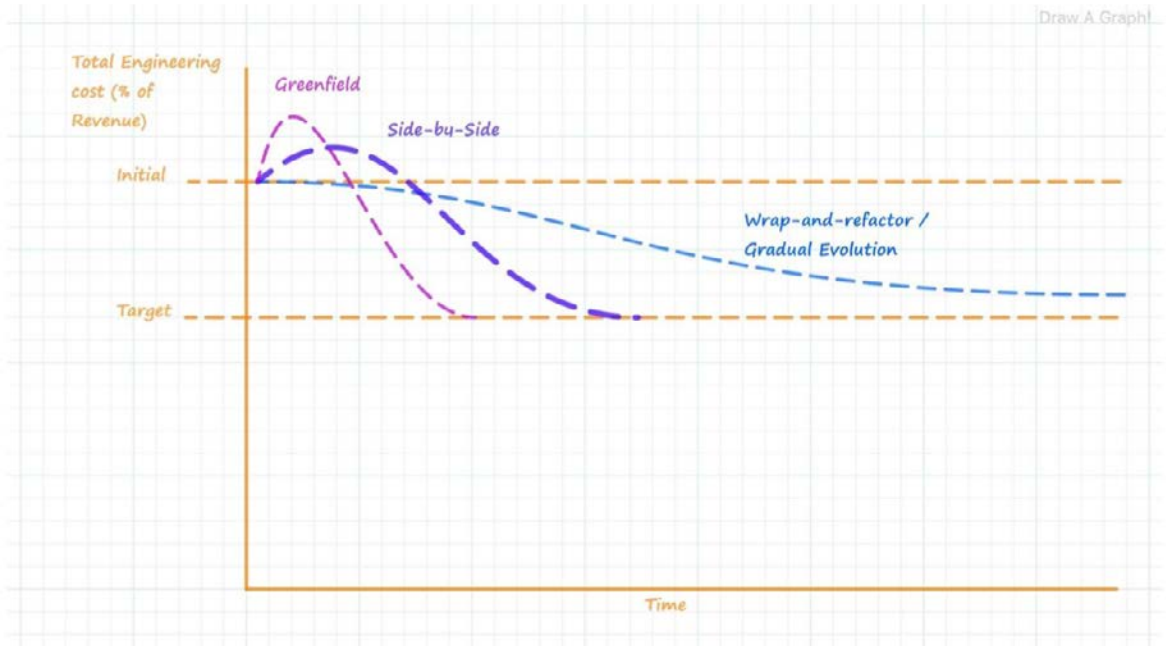

Figure 6: Side-by-side cost curve

The Side-by-Side approach has the most variations of any technical transformation strategy, and we will discuss some of the most important ones.

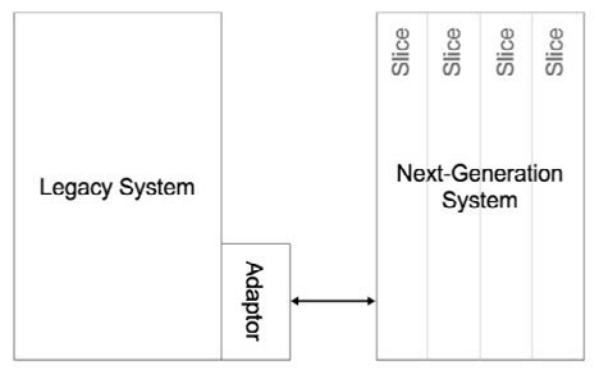

Figure 7: Side-by-Side approach

The Side-by-Side approach is a way of reaping many of the benefits of the Greenfield approach while mitigating its time-to-value issues. It does this by deploying a Greenfield "NextGen" system incrementally so it can start driving business value before reaching full feature parity with the legacy system. To make this happen, a subset or "vertical slice" of the final system is built in a "pure" NextGen architecture and deployed in parallel ("Side-by-Side") with the legacy system. Depending on the details and the amount of integration involved, this can be a highly effective approach that drastically shrinks the "time to value" for key elements of the NextGen system.

The ultimate end result of a Side-by-Side implementation approach can be the same as achieved by taking a Greenfield approach. In fact, when implementing this approach, we like to start out by architecting the ideal Greenfield system, and then working backwards to determine the staging of work into production.

The primary difference between the Side-by-Side approach and Greenfield initiative is that the Side-by-Side system is implemented in "slices;" each slice is put into production before the entire system is completed and can potentially generate business value. These NextGen slices provide part of the functionality to the end user, and the legacy system provides the rest. The Legacy system and the evolving NextGen systems are coordinated by means of an adaptor layer on the legacy that either leverages already-existing APIs and external interfaces (uncommon, but great if it happens), or that is created as part of the migration.

There are two primary roadmap strategies that can both benefit from a Side-by-Side approach. One is to use the "NextGen" system to deploy new features not currently provided by the legacy. This can be a very useful "escape valve" to rapidly address pent-up demand or competitive threats which have not been practical to address through the system. The key time-to-value constraint tends to be the creation of the adaptor layer, but if APIs already exist or if legacy integration can be light-weight, then you can usually deploy new functionality rapidly (weeks or single-digit months).

The other roadmap approach is aimed at retiring the Legacy as soon as possible. In this plan, already-existing legacy functionality is re-implemented in NextGen, and decommissioned in the legacy system as you go along. A combination of both roadmap approaches is also common: implementing an enhanced version of current functionality in NextGen, for example.

In the Side-by-Side approach, you progress toward your blank-sheet-of-paper / ultimate Greenfield architecture over time. Instead of implementing it directly as you would in the Greenfield approach, you instead stage the implementation and deploy the new system "use-case-by-use-case," for example.

While the ultimate end result can be the same as you would achieve with a true Greenfield system, the mandate to bring the system to a deployable state multiple times does impose overhead. Among other things, this makes the integral of the total level of effort over time higher for the Side-by-Side approach than it would be for an identical system crafted under a pure Greenfield approach. The need to create, enhance, and maintain a throw-away "adaptor" that interfaces with the legacy also adds to the overhead.

If you're not careful, the process of staged development and deployment, together with the requirements of continuing legacy co-existence, can compromise the final result architecturally and in terms of the end user experience. It's easy to get into a mode of short-term compromise to achieve an intermediate deliverable even when it has a negative effect on the final product. The Side-by-Side approach requires more discipline than Greenfield to achieve an uncompromised end result.

On the other hand, Side-by-Side not only offers a more rapid time to value, it also delivers early market validation and feedback as a clear benefit. This approach also has value as a process improvement forcing function and as a user and technical testing vehicle. These benefits and others come from learning

to deploy "for real" more often, and putting your work product in front of real users. It also supports a "lean" iterative approach to product development, allowing rapid improvements based on real-life user feedback.

However, as they say, "there ain't no such thing as a free lunch". The Side-by-Side approach introduces a new set of downsides of its own. These include:

- Moderate inherent organization transformation benefits. In terms of its inherent ability to transform your current staff and organization, the "Side-by-Side" approach is somewhat better than the Greenfield approach, but not much. More people from the Legacy team will generally be involved with the new technologies because of the need for integration of the NextGen system with your legacy system. This integration activity in itself provides a bridge between the old and the new skill sets. On the other hand, the bulk of the NextGen work is still done largely independently of the legacy system, leading to some teams with "old" skill sets and other teams with "new" skill sets. Unless consciously mitigated—or unless bringing the current staff into the future is not a consideration (because, for example, the legacy sustaining work is done by a 3rd party that will be offloaded or phased out)—then you will end up with separate "Legacy" and "modern" organizations.

- The cost of integrating with the legacy system. To work effectively in a "Side-by-Side" configuration, work must be done to integrate the "new" system with the "old" one. Whether this integration is significant or minimal, it is throw-away code which will no longer be required when the NextGen system is completed and the legacy system is deprecated. During the time the integration layer exists, it adds complexity and cost to your overall system since it must also be managed and maintained.

- Overhead of building "vertical slices" and integrating each of them with the legacy. Instead of building your next-generation Greenfield system in the most natural and efficient way, you must stage development so that you can periodically stand up fully self-contained collections of features that are each integrated with your legacy. As we've mentioned, this is not entirely a bad thing. Deploying production-quality subsets of your NextGen feature-set at regular intervals is the strongest validation possible of your new system. You are also "hedging your bets," since functionality can be provided by the legacy until the "NextGen" system is truly ready.

On the other hand, this imperative to deploy slices in production also imposes technical and schedule constraints which complicate product and program management, lengthen the overall development cycle, and add cost. In particular, the need to integrate with the legacy at each stage of NextGen development slows implementation speed relative to a pure Greenfield approach and results in additional throw-away work at each stage.

- **Requirements and reverse engineering time and cost.** Like the Greenfield approach, the Side-by-Side approach requires that you describe the features and functionality of the system you want. You also have the added cost of reverse engineering and describing the system you have today—at least to the extent you need to do so in order to integrate with it. The forward-looking product definition activity is less risky than in a Greenfield approach, since a Side-by-Side approach retains the legacy system to fall back on for functionality you may overlook.

  However, the ultimate goal remains to replace what you have today with the system you want for the future. While the transformation is clearly an opportunity for improvement, it also requires putting in the effort to figure out what it is you really do want at a considerable level of detail. This takes time, effort, and sometimes new skill sets, since the future will, in almost every case, not be the same as the past. On top of that, you have the cost of understanding your existing legacy system in more depth than you need to with the Greenfield approach, because you still need to integrate with your legacy for some time.

- **Perhaps most importantly, your overall system complexity increases until you finish the "new" system, and deprecate the old one.** If you don't finish the migration for any reason, you now have two systems sitting side by side using different technologies, plus an integration layer that you need to maintain. Some system architectures we have reviewed have multiple layers of never-completed next-generation initiatives delivering subsets of features in a mosaic of technologies and architectural approaches. When this happens, it can leave the system in a state that is harder to maintain than the original legacy.

Because of the integration and productization work that must be done to reach the same end-state, a Side-by-Side approach takes more effort overall than a Greenfield approach. This generally manifests as a longer project time,

Side-by-Side Approach
Integration at the Glass

As we mentioned in the previous section, there are likely more variants of the Side-by-Side transformation approach than any other. Now let's consider the "Integration at the Glass" variation of the Side-by-Side transformation strategy.

The lightest form of Side-by-Side integration that tends to be useful is "integration at the glass". In this paradigm, the user interface is implemented over a mixture of legacy code and a partially-implemented NextGen system. If done well, the UI is seamless and looks to the end user like it's deployed on top of a single back-end system—hence the term integration "at the glass" (meaning "on the display device" / "at the interface").

In this migration paradigm, more and more of the UI is converted to a NextGen system implementation over time until ultimately the legacy back-end is entirely replaced. At that time, the UI can be modernized and an upgrade announced with fanfare—though in fact the migration has been happening progressively "behind the scenes" for some time.

While it's easiest to describe this approach in terms of a user interface display (the "glass"), the same approach works for any form of external interaction that is "discrete"—that is, loosely coupled to other system functionality. For example, an interaction with an external system that gathers information and independently persists it could be refactored in the same way. You would do this by keeping the interaction protocol and interfaces with the external system the same while you change the underlying implementation. After changing the implementation, you then have the flexibility to rapidly add or support new interaction protocols, or negotiate a change to your current protocol if desired.

In terms of UI, systems lending themselves to an "integration at the glass" approach generally have a "page-based" or "form-based" interaction paradigm, where a subset of the "pages" of the legacy system UIs are, essentially, discrete applications that interact very little with each other. Old-style, database-centric CRUD applications, where each page basically writes to a different RDBMS database table with minimal coupling and back-end business logic, are good candidates. Even if the front-end (client) contains business logic to the extent each page is "stand-alone", this approach can still be useful (though of course

you will want to migrate the business logic out of the UI and into server-side logic in your new system). In such an integration-at-the-glass scenario, you can go page-by-page modernizing or creating "back-end" functionality that services a given "page" or set of "pages." The back-end functionality can use your next-generation system paradigm, while the front-end implementation may use new technologies and a modern Web, Mobile, or other client architecture as well. So long as it's done "page by page," this is an instance of the "integration at the glass" strategy.

Because it's often difficult or wasteful in terms of engineering effort to try to change the legacy UI code, it's generally simplest keep the UI interaction model and look-and-feel as-is until the whole system can be updated. This means retaining the legacy system's look-and-feel in your new UI code and temporarily implementing this retro look using modern UI technologies as you go. That way, to the end user it appears nothing has changed, even though a subset of the "pages" has actually been re-implemented in your NextGen front- and back-end architecture.

To the extent that the functionality is truly discrete ("page-by-page"), the integration of these new "slices" on the server side also tends to be straightforward. Because modern UIs, implemented correctly, are far easier to change than your old legacy UIs, your client's look-and-feel can be updated rapidly once the behind-the-scenes system upgrade is complete.

Systems lending themselves to this lightweight integration approach tend to be older and algorithmically straightforward—for example, systems oriented primarily around data storage and retrieval. These types of systems still exist in large numbers, sometimes deployed at massive scale, and this approach can and does work for them. Where there is more logical or algorithmic complexity, however—for example, where the data on different "pages" interacts in a visible and significant way—a deeper integration between the NextGen system and legacy may be needed. This means a more elaborate migration strategy is required than "integration at the glass,"such as one of the other Side-by-Side approaches.

Side-by-Side Approach
Integration at the Glass
Case Study

We previously described the "Integration at the Glass" variant of the Side-by-Side transformation approach. This approach is useful in applications where—unlike a spreadsheet or a game—the client application consists of discrete "pages" that are largely independent of each other. Even today, with the rich user interaction paradigms now available to us, this type of page-based application UI is still relatively common. In particular, where the primary business activities concern data entry, retrieval, and reporting, which is still a common use case, the page-based user interaction paradigm is widely applied.

Applications that have "sections" of discrete functionality—for example, menus that take the user to different discrete accounting functions like AR, AP, or GL—can also be candidates for this approach. While not literally "page-by-page", such a transformation can be done "subsection by subsection" using a similar transformation paradigm.

The client in our example is an established, mid-sized, software-enabled services company providing specialized B2B services to a small set of vertical markets. While they have an in-house development team, the client did not consider themselves a software or technology company; instead their primary business was providing specialized services to other companies. They used their in-house developed software system as a means to the end of providing those services, primarily by tracking and coordinating their customer and service activities.

At a high level, the software's primary use cases centered around data entry, data retrieval, ranking, grading, filtering, and reporting. The software system offered "pages" to input, view, and update the required information, with each "page" addressing a more-or-less discrete service function.

Based on their role, each individual user of the system would, in general, focus on one or a small number of pages for their day-to-day business activities that involved this system. Certain "pages" of the system were used exclusively by the company's internal employees, while others were used primarily by external customer; though, to a lesser extent, they were sometimes used by internal employees as well, especially for support purposes.

The company started receiving increasingly significant complaints from its customers about the usability of the pages that were accessed externally. The system's UI paradigm was frozen in place many years ago, and from a look-and-feel, as well as a usability, perspective, it was no longer seen as acceptable. While the primary external use was still on desktops, support for mobile devices was becoming an "ask" and there was no economically practical way to deliver a responsive design given the current system architecture.

While the employee-facing pages had the same usability issues, it was not a primary concern to the client because it could train its own employees. The business problem with customer usability was that competitors with more modern interfaces were seen as more usable, especially by external users who only accessed the system occasionally. Some of those external users were low-level, hourly employees in positions with a high turnover rate. This meant that, in order to mitigate the competitive threat by using training and support of those hourly external employees, the client's costs were becoming uncontrollable. In addition, the client had acquired a number of competitors with better interfaces.

It was politically challenging, or even impossible, to get customers of the acquired systems to switch from their modern UI to a system whose usability was worse. None of the acquired systems had the scope to replace the full legacy, so the legacy could not be deprecated in favor of one of them. The net effect was that the individually acquired "niche" applications needed to stay in operation to keep their inherited customers happy, which further added to costs.

Unfortunately, because the system architecture evolved over time for the client's legacy, it was not a simple matter to modify the look-and-feel of a given page in any but the most superficial ways (e.g. color scheme). Like many systems, the software had grown organically over time. The original authors implemented the system logic primarily as a set of stored procedures ("SPROCs"), and the number of SPROCs had grown to many hundreds. With some template-based UI generation layered on top, page generation was coupled directly to one or more stored procedures which did data retrieval, data formatting and storage, and applied necessary business logic.

As it grew, the "View" layer (essentially the UI) of the system had expanded to contain significant business logic. Also, the stored procedures contained embedded assumptions about how the UI worked, up to and including directly generating portions of the UI themselves. In other words, there was no separation of concerns between display, logic, and data access; they were inextricably woven together.

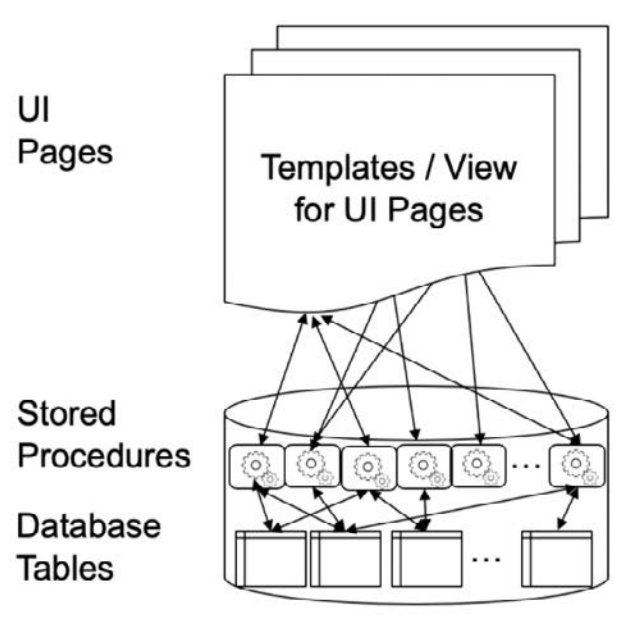

Figure 8: Original system architecture, "Integration at the Glass" use case

Our initial proposal was to incrementally build a second, modern system parallel to the first ("Side-by-Side") and integrated "at the glass". The (undocumented) business logic that the legacy system implemented in both the UI "View" layer and in the stored procedures would be reverse engineered and re-implemented in a modern business logic layer. From an end user perspective, system operation would be unaffected during the migration, even though some pages would be supplied by the "modern" NextGen system, while others would still be supplied by the legacy system.

The only change the end users would notice was that some pages would now look different and better than they did before. These UI improvements and the migration of business logic would continue incrementally until all the pages had been converted; then the old system could be deprecated.

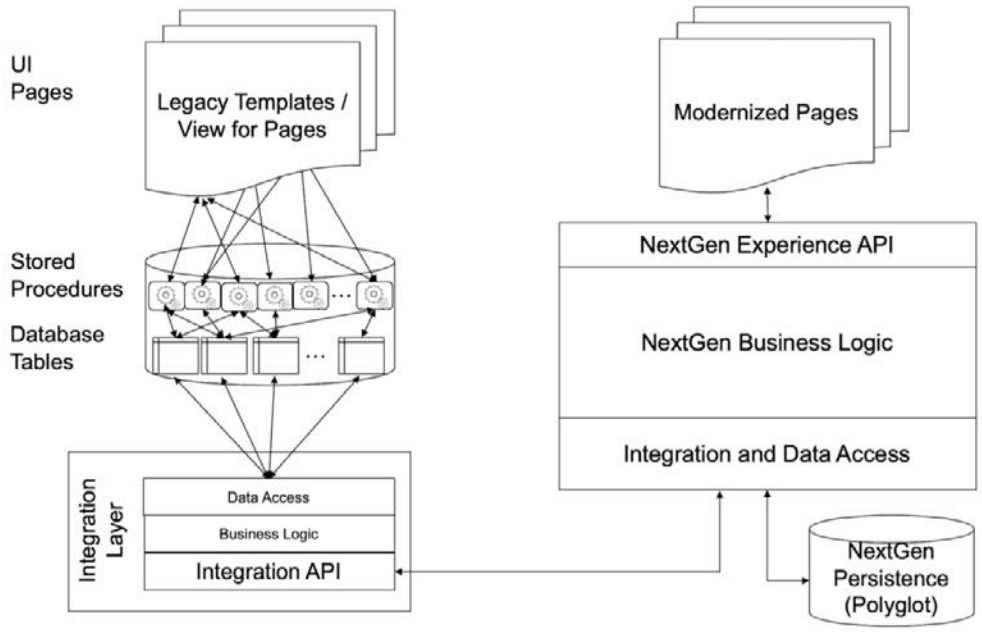

Figure 9: Original migration proposal for "Integration at the Glass" use case

The client rejected the progressive reimplementation approach, however, stating that the focus should be solely on enabling improvements to the user experience by decoupling pages from the stored procedures. The sole goal was to get to a greatly improved UI, not to provide deep refactoring of the whole system.

We acknowledged that, per page, the up-front burn rate would indeed be higher to do deep refactoring. This is because more work would need to be done to port a given "page" to the new system, including understanding and re-implementing the logic behind the page that currently lived in both the View layer and the SPROCs. However, we pointed out that the overall total cost of migrating to a modern system—fully or partially—would be lower if the logic was re-implemented at the same time the UI was refactored. Understanding and implementing new business logic needed to be done to some extent to

understand the page behavior and to retire the business logic from the legacy UI View layer. All agreed this work must be done in any scenario, before the UI could be modernized.

We pointed out that it would be much cheaper overall to completely extract and re-implement the business logic from the SPROCs now, than it would be to first refactor the SPROCs to remove UI dependencies, then go back and complete a full refactoring later. Staging the SPROC refactoring work sequentially—first refactoring the SPROCs to remove UI dependencies and then later going back to fully re-implement and retire them—would be more work overall and take longer than doing both at the same time.

Because of the client's financial situation, however, they decided not to make any incremental investment beyond the bare-bones cost of improving the UI for the important external pages. They acknowledged that, while the overall cost and time would be higher to get to a fully modernized system, keeping the incremental investment as low as possible in the near-term was their key concern. Within this constraint, we therefore proposed and worked with the client to execute the following scheme.

Note that all of these steps were executed against the running system, while it was in production. They caused no downtime, no changed behavior, and no negative impact on the clients. All the clients perceived was that page UIs were selectively being improved.

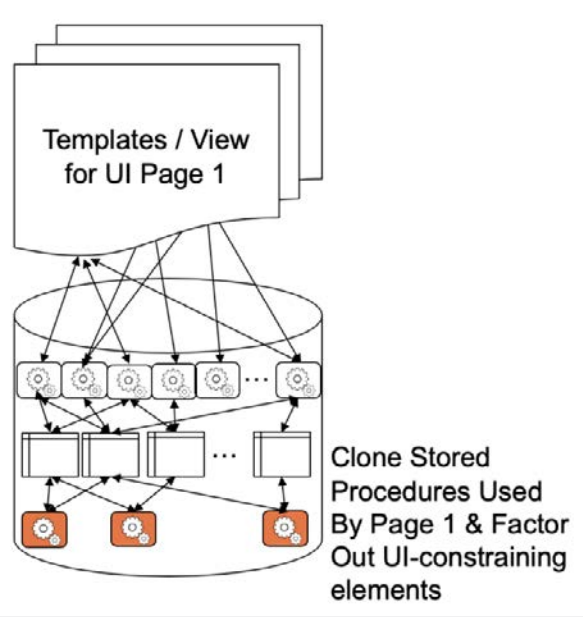

Figure 10: "Integration at the Glass" use case, Step 1: clone stored procedures

1. We proceeded page-by-page, and worked with the client to identify the key external pages and the order in which to proceed. We set up a process to reverse-engineer each page, first from a functional and behavioral standpoint (for which the client supplied the expertise), and then from an implementation standpoint. We also identified the stored procedures and any other logic that was required to generate the legacy page. The required SPROCs were "cloned"—that is, copied and tagged with an indicative prefix. The cloned SPROCs were then inspected by hand and modified to remove any UI assumptions or UI generation code that would stop those SPROCs from being used against a newly implemented page.

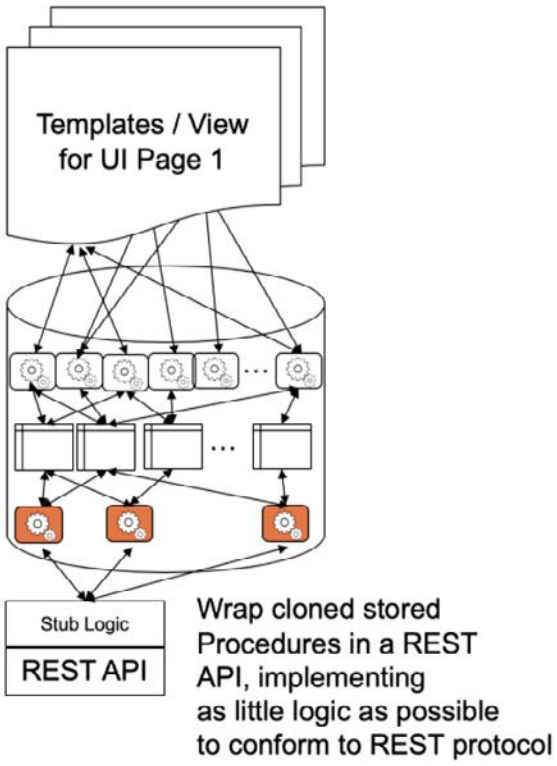

Figure 11: "Integration at the Glass" use case, Step 2: wrap cloned SPROCs in a REST API

2. Once the relevant SPROCs had been cloned, and the UI dependencies removed from them, we wrapped them in a REST API, introducing a minimal amount of business logic to support the REST paradigm. REST is a stateless paradigm, meaning that there is no notion of a persistent session between

a client and a server. Instead, each API call from the client to the server, or vice-versa, passes the context required to give the end user the impression they are continuously connected to the server, when actually each call is a uniquely handled event. Implementing this protocol can require a small amount of logic, and this was done external to the SPROCs. The goal on all sides was to keep modifications of the SPROCs to a minimum since they would be discarded at some point in the future anyway.

The REST paradigm became popular as a way to support massive scale; in particular mobile applications. That scale was not really required in this case, as the client's system is "enterprise scale" (supporting tens of thousands of users), not "web scale" (tens of millions or more). However, modern UIs are based around the REST paradigm, and we wanted to be able to use the modern paradigms freely; this is the main reason for the wrapper we created. A secondary reason is that once such an API is in place, other clients—for example, mobile clients—can readily utilize the same connection point to the server. This gives enormous flexibility, and will allow the client to stay competitive even if they need to move beyond a modern responsive Web UI

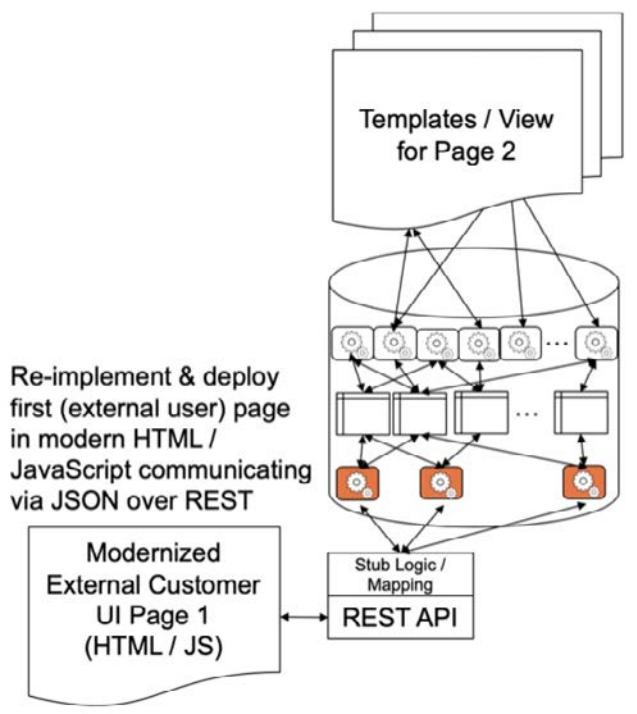

Figure 12: "Integration at the Glass" use case, Step 3: replace the legacy first page with a modern UI

3. Next—actually in parallel with the previous steps—we replaced the legacy "Page 1" by creating a modern responsive Web UI, using HTML and modern JavaScript frameworks. Even though the pages are largely independent, there are some common contextual and stylistic elements, including navigation bars, menus, external customer branding, logged in user name, and so on. These are readily supported by modern frameworks and by the REST paradigm.

   A design system was developed so that the "New" pages were significantly improved, especially in terms of being responsive (in particular, working properly on the browsers of mobile devices). However, the page designs were still kept similar enough to the legacy application that the contrast between pages which co-existed Side-by-Side would not be jarring.

   In parallel, we also reverse-engineered and refactored the business logic contained in the "View" layer that, in the legacy system, had been used to generate the page. This business logic was factored out of the page itself and re-implemented in the "stub logic / mapping" layer in Java. The APIs were named so as to reflect their business function.

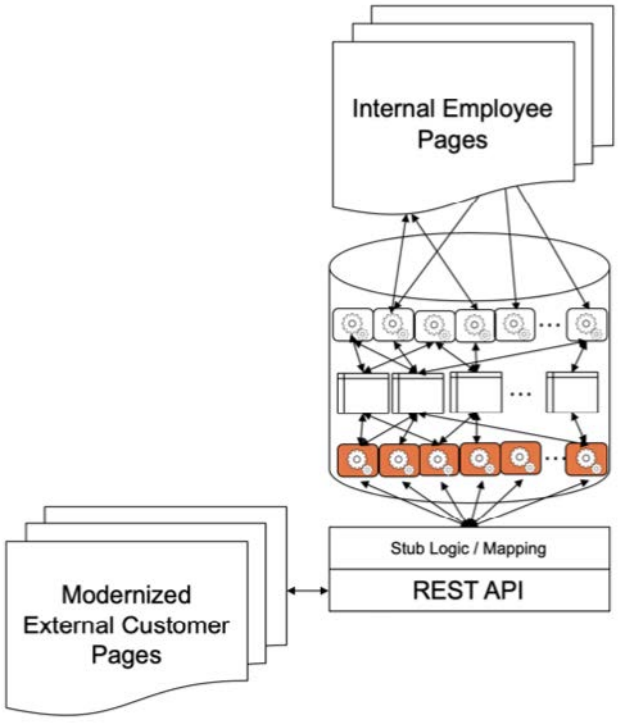

Figure 13: "Integration at the Glass" use case, Step 3: repeat until all external pages have been replaced

4. This process of reverse-engineering pages and SPROCs, cloning and refactoring UI logic out of SPROCs, re-implementing "View"-level business logic into an explicit business logic layer, wrapping the whole in REST APIs, and implementing a modern replacement UI was repeated until all key external customer pages had been upgraded. This process took approximately 3 years from start to finish because of the number of pages and SPROCs, and the amount of reverse engineering, design, and refactoring required.

If the client had accepted our original proposal for a complete refactoring, it probably would have taken roughly the same amount of time. The cost would have been higher because of the added refactoring work—at a guess 50% higher—but the modernization of the entire system would have been done. As it stands, the system is only partially modernized on the UI side: the "external" pages.

On the platform side, business logic remains centered around slightly improved SPROCs (plus a thin business-logic layer containing old "View" logic) rather than in modern cloud-native technologies. Except for the refactored UI pages (and associated APIs), the rest of the system remains almost as hard to maintain and enhance as they were at the outset of the project.

But the system is where it is. On the positive side, the client's originally stated goal of an improved UI for the key external-facing pages has been met. To take the next step to full refactoring, given the current state of the system, a logical way to proceed from here would be to next deprecate the "cloned" SPROCs.

We would do this by re-implementing the SPROC functionality outside the database and in a modern "business logic" layer built on top of a data access layer that provides abstraction from storage details. The data access layer would initially talk directly to the existing legacy database table structure. This allows the existing legacy SPROCs to continue to function and serve the needs of the internal pages.

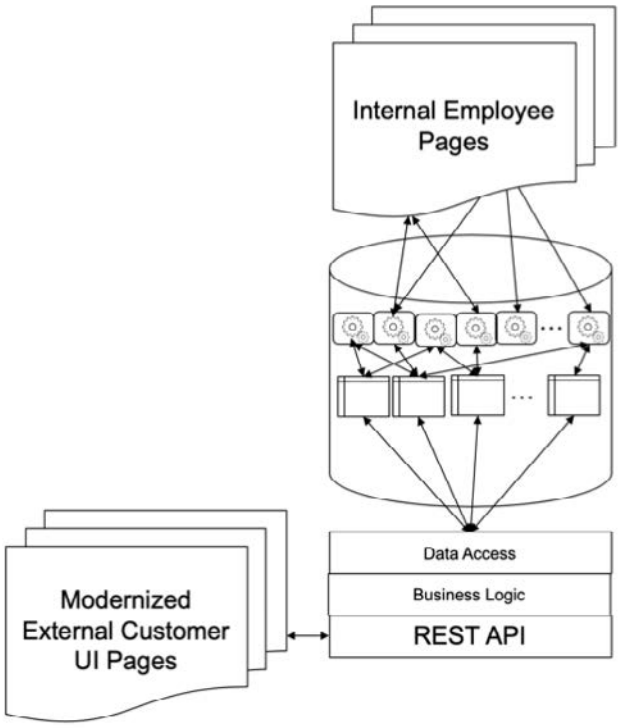

Figure 14: "Integration at the Glass" use case: deprecate cloned SPROCs

As a further extension of the "integration at the glass" paradigm, this migration could also be done page-by-page, by first re-implementing the functionality of the SPROCS that serviced "Page 1" into the business logic and data access layers. There would be no need to clone these SPROCs, since they would be re-implemented from scratch in the business logic and data access layers. Once the re-implemented code is in place and attached to the API and a modernized page deployed, the cloned SPROCs that only serviced page 1 are no longer accessed and could therefore be deprecated.

Those SPROCs that serviced page 1 and other pages would need to be retained until there were no longer any pages that needed them and then they could be deprecated, too. This process would be repeated page-by-page, until all the remaining SPROCs were replaced by modern business logic and data access layers external to the database. Again, this could be done against the production system, and clients would not notice any change at all.

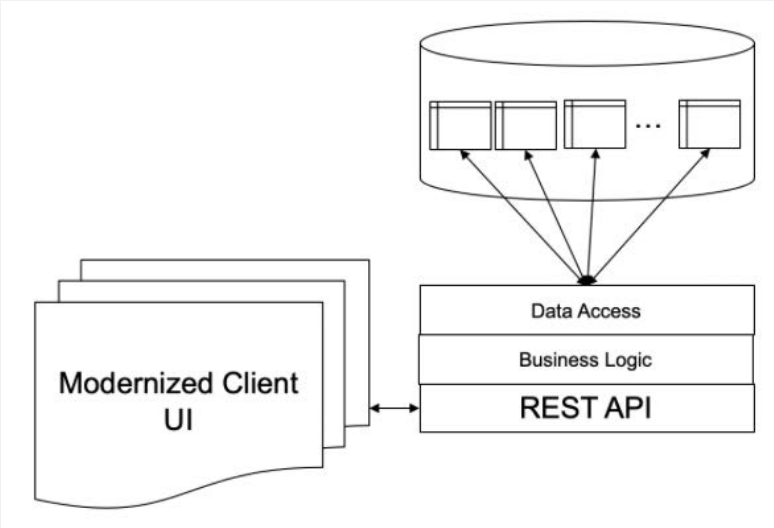

Figure 15: "Integration at the Glass" use case: end state; deprecate all SPROCs, fully modernized UI

At the end of this process, the current database tables would remain; however, given the abstraction provided by the data access layer, the table structure could now be refactored with no impact on the rest of the system. Also, a full UI modernization that included migration away from a page-based paradigm (if desired) could now be performed with no impact on the back-end system.

Side-by-Side Approach
Extend and Replace

In the previous section, we discussed the "Integration at the Glass" variant of the Side-by-Side transformation approach. While that approach is simple and can be useful in some situations, it is not universally applicable to a wide variety of legacy platforms.

A more generally applicable transformation strategy is one that we call "Extend and Replace". In this approach, a "vertical slice" of the Next Generation system is implemented and deployed in production. This "slice" is integrated with the legacy system, and either adds new features to the legacy ("extend"), or re-implements current features in the new architecture ("replace"). Over time, legacy features are migrated to the next-generation system until the NextGen system entirely replaces the legacy.

The essence of this approach is that we create a new system Side-by-Side with the existing one, which will ultimately replace it. In contrast to the Greenfield approach, we build and deploy this new system incrementally, feature-by-feature or use case-by-use case, until the "old" system has been replaced. This approach avoids issues with the legacy code base by not changing it. The current code base is left as-is to the fullest degree possible, until it can be thrown away.

There are actually two variations to "extend and replace", but they are so similar in an engineering sense that we will discuss them both here. In one variant, the Side-by-Side system adds new features or functionality to an existing system. In other words, the new system delivers something that the legacy system does not. In the other variant, the new Side-by-Side system replaces a feature or function that is already fully or partially performed by the legacy system.

In the feature-addition case, new features can either be integrated with the existing functionality in some way (a new menu item, for example), or introduced as a new stand-alone capability with its own UI and entry points. Both variants are common. In the case where existing features are replaced, the old

implementation is deprecated, and requests are sent to the new system instead. This can often be done without significant change to programmatic interfaces (API) or to the look-and-feel of the existing UI, providing a smooth migration path "while deployed".

These two variations generally go hand-in-hand. In a transformation situation, the goal is to replace the existing legacy system with a new and better one that supports new business and revenue models; adds new features; is more capable, better architected, and more scalable; is easily enhanced, cloud-native, and mobile-first. In other words, it is an "upgrade" in these and all other ways.

Given that the end goal is an overall improved next-generation system, the decision to first deploy added features or first improve existing ones is a matter of prioritizing and staging the work, as well as sometimes style. The key, as always, is to take the holistic business picture into account, both internal and external, in deciding what to do first. That way, you can make the flexibility offered by the "extend and replace" approach work for you.

Side-by-Side Approach
Extend and Replace Case Study

In the last section, we talked about the "extend and replace" variant of the "Side-by-Side" transformation approach. To make this discussion more concrete, let's look at a real-life example.

One client brought to the table a tightly-coupled system which had grown organically over several decades (literally), culminating in 30 million lines of difficult-to-maintain legacy code—mostly in C, but with a mix of other languages as well. Critical portions of the core system had been written so long ago that many of its original developers had not only retired, but died. There was no documentation, as is common, and also literally no one left alive who understood the details of various critical elements of the core system.

The system implementation was highly monolithic and tightly coupled, to the extent that new feature development was essentially paralyzed. Even a small and careful code change made anywhere in the system often resulted in unpredictable and apparently random problems surfacing somewhere else. Given the mission-critical nature of the system to its end users, the fragility of the code base meant that even the smallest change required a complete system retest and consequent bug fix cycle.

Since each bug fix had a high likelihood of introducing other problems, a single initial change could trigger a cycle of fixes and breakage that consumed weeks or even months. Compounding this was the long onboarding period required to train new developers, the high turnover rate among frustrated new developers, and a core workforce nearing retirement age.

The net effect was a feature set that had remained largely static except for delivery of the most urgent fixes to the most irate and vocal large customers, accompanied by increasing rates of customer dissatisfaction. Still, the company's niche was specialized and conservative enough that they enjoyed

a nearly 70% market share in their space. However, new competitors were beginning to enter the client's space, using our client's system as a "system of record" and building new features on top of it. While our client's maintenance revenue stream continued, they were losing new software purchase dollars to these nimble upstarts. These competitors also did not have the overhead of building or maintaining the "core" system, since they could leverage that of our client, which freed them to focus on value-added features the client's system could not be enhanced to perform.

The client's system was deployed on-premises at their enterprise customers and included a tightly-coupled "fat" desktop UI which contained significant business logic. The UI code was tightly integrated with the system "back-end", with only a few front-end features being supported by a back-end API. Because considerable business logic existed in the front-end, and there was no simple "page-based" user interaction model, an "integration at the glass" approach was not feasible.

While the client wanted to migrate to a completely modern system architecture, they faced an immediate competitive threat because of the emerging, nimble competition. Most acutely, they had no mobile or cloud strategy, and this was becoming a major concern for their customers. Indeed, given the way the fat-client was closely integrated with an on-premises back-end system, with both containing undocumented and non-understood business logic, there was no simple path to decouple them.

While a "remote desktop" / "terminal services" approach would have enabled the client to host their current fat-client code base on the cloud, their current UI structure would not neutralize the competitive threat for mobile devices. Neither would such a "lift and shift" approach gain them any economies of scale, nor allow more rapid addition of new features.

Given these goals and constraints, we proposed an "Extend and Replace" migration strategy that would enable a mobile-native interface to be developed rapidly (in about 4 months), as well as give the client a credible direction for future cloud migration that they could announce to their increasingly impatient customers. This approach was, intentionally, the same as that being used by

their successful new competitors: Use the existing legacy system as a "system of record", and build new functionality "on top" that was not encumbered by the legacy architecture. While we took what was, in essence, the same approach, our client had the advantage over their competition in that they could access features of their system that were not exposed to external developers if we could figure out a way to externalize them.

The key elements of the first phase of the strategy we came up with were:

- Clone the client's existing, tightly-coupled UI, and modify it into a "headless agent" which could be deployed on-premises at their enterprise clients with the rest of the on-premises system. In this strategy, APIs would be developed on top of the existing client code—that is, replacing the UI functionality on top of the existing UI business logic. No effort would be made to refactor the UI business logic or to decouple the existing client from the existing back-end functionality, since this code would ultimately be discarded.

- Stand up the nucleus of a next-generation system in the cloud (AWS in this case), implement the functionality required to route to, and interact with, the appropriate on-premises agent, and provide the required features through a modern JSON over REST stateless API. These Cloud APIs, at first, were simple pass-through mappings to the functionality deployed on-premises, with only the logic and persistence required to handle session state and make the Cloud API calls stateless REST APIs. At first, these Cloud APIs were simple pass-through mappings to the functionality deployed on-premises, and only had the logic and persistence required to handle session state and make the Cloud API calls stateless REST APIs.

- Implement a modern mobile native application against the Cloud-based REST APIs.

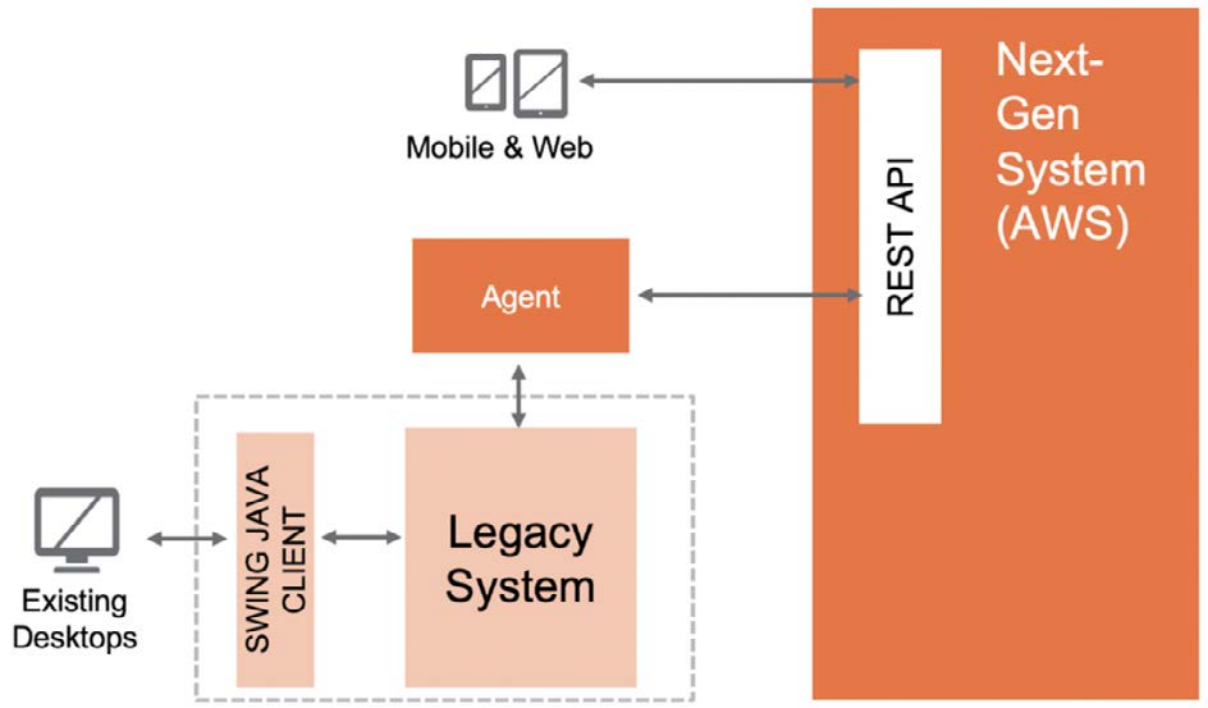

Figure 16: Example of "Extend and Replace" approach

In the above diagram, the "Agent" is the throw-away integration component that enables remote use of the on-premises legacy system. As legacy functionality was re-factored and re-implemented in the NextGen, Cloud-native architecture, a call out to the on-premises Agent was either no longer required, or else served solely to update the persisted data in the legacy system so the remaining legacy functionality would continue to work.

These calls could be made via the existing business logic, or the Agent may have to write directly to the database, where that was strictly necessary. Over time, the legacy system would be deprecated entirely, and the "Fat," on-premises, desktop client replaced with a Web client connecting directly to the NextGen system in the cloud.

Note that, as features are added to the NextGen system, the integration component ("Agent") must be actively maintained throughout. The legacy system is, by design, changed as little as possible until it is disposed of entirely.

The essence of this approach is to achieve a quick win (mobile functionality in this case), while buying time for a deeper refactoring. Total migration to the NextGen system is a major effort which requires significant reverse engineering of the current system and the ability to define product requirements for the "as-desired" next-generation system.

In the next section, we'll discuss another variant of the Side-by-Side transformation strategy that we call "Present-Forward" / "Future-Back."

Side-by-Side Approach
Present-Forward / Future-Back

In the previous section, we discussed the "Extend and Replace" variant of the Side-by-Side transformation strategy. Now let's discuss another variant we call "Present-forward" / "Future-back". This approach is a hybrid of the Gradual Evolution and the Greenfield approach, with the twist that the two are constructed simultaneously.

The analogy is to a major physical construction project, such as a bridge, tunnel, or railroad. In these types of construction projects,one crew often begins work at one end of the structure, while another crew starts on the other. The two crews independently work toward each other until, by good planning, they meet at an intermediate point.

The advantages of this approach include using parallel work to get done much faster, while also achieving some potential cost reduction in shared overhead functions such as management, logistics, cost of materials (due to bulk purchases or shared component-level construction costs), and so on. The US Golden Gate Bridge, the Australia Sydney Harbor Bridge, the UK / EU Channel Tunnel and the US Transcontinental Railroad are just a few prominent examples of where this construction technique has been successfully utilized.

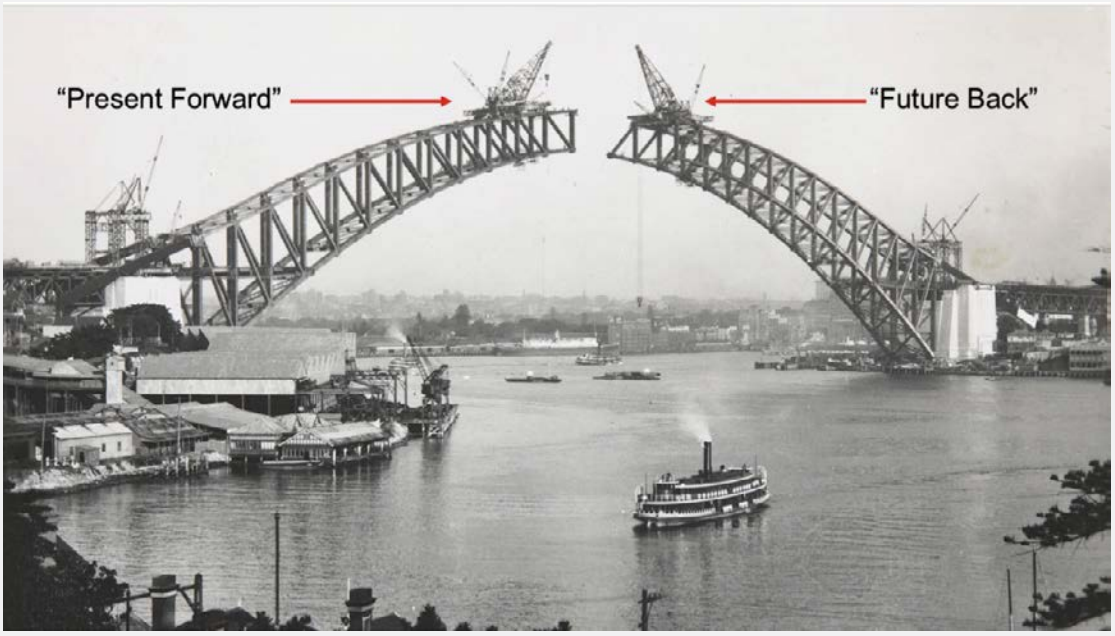

Figure 17: Present-forward / Future-back approach (image courtesy National Museum of Australia, nma.gov.au)

In software, this paradigm is played out against a timeline, rather than spatially. One "side" of the work is the current state of the system. This present state is evolved, using a Gradual Evolution approach, into the components of a next-generation system. The team that starts with the present state of the system is called the "Present Forward" team since they evolve the current system forward into the next generation.

The second "endpoint" of the work is the aspirational future state of the system. A second team begins implementing a subset of the next-generation system using a Greenfield approach. This team, because it starts from the future system and brings it into present reality, is called the "Future Back" team.

The goal of both teams is to "meet in the middle" in a single system that includes both present-forward, evolved components, as well as freshly created Greenfield / future back components. This requires significant planning and coordination, but where the end-state architecture is sufficiently modular, it will support this approach.

A Present-forward / Future-back strategy has several advantages:

- Because the current platform is working at all times while it is being evolved, risk mitigation is built into the approach. You have the option to fall back onto the Present-forward branch at any time.

- Whichever approach turns out to be fastest can naturally be allowed to take on the lion's share of the work. There's no reason (in software) to meet precisely in the middle to get the job done. If the future-back implementation moves faster than the present-forward approach, then more of the work can be done by the future-back team. On the other hand, if the present-forward approach moves faster, then that team can build proportionately more of the final system. The key is implementing your desired system end-to-end ASAP. The means can vary depending on the actual speed achieved by the teams in practice.

- Teams can have diverse skill sets, initially. The present-forward team needs to be skilled in current-generation technologies, while the future-back team needs expertise in the technologies you will use in your next-generation systems. With this approach, both skill sets can coexist harmoniously and the skill sets of both teams are highly leveraged. Over time, the present-forward team will need to learn the next-generation technologies, and the future-back team will need to understand the domain. However, this learning can be staged in a very natural way within this approach.

- The transformation value of this approach is high. With current-generation and next-generation teams working (virtually) Side-by-Side toward a common goal, morale and knowledge transfer are naturally high. Your current-generation team's skills are highly valued and leveraged in this approach, while the next-generation team is able to move at full speed. Knowledge flows both ways, allowing the next-generation team to become skilled in your domain, and the current-generation team to learn the new technologies.

- Provided the current system has the features you need to stay in operation, you need not integrate the future-back code base with the present-forward system until you are ready for them to "meet" and fully merge. This reduces and defers the integration cost until the time the components are intermingled—which, if done in a highly componentized architecture, can be done very smoothly.

People sometimes choose the present-forward / future back approach when the investment in the current system is such that it can't be retired. This may be due to internal or external "political" considerations, because of business concerns about a financial write-down that may result in retiring current code, because domain or other knowledge has been lost and the safer course appears to be encapsulating and preserving a subset of the actual legacy implementation, or for other reasons.

While it offers a number of advantages, the present-forward / future back approach inherits most of the downsides of the other Side-by-Side migration strategies, as well as adding some new ones. These unique drawbacks include:

- The present-forward / future-back strategy does not inherently make it easier to add extensions to the functionality of the existing system while it is under development. There is nothing in the approach that prevents you from adding new functionality on the "present-forward" side—but nothing that makes it easier to do this, either, at least until refactoring is well advanced. If rapid extensions and pivots are required, they are supported more organically by other approaches—for example, the "extend and replace" approach (see example above).

- In most scenarios, you see limited or zero business value from the "future-back" work until it is integrated with the "present-forward" work. In some situations, this can be mitigated by producing intermediate deliverables in the form of "vertical slices" against the future-back work, using an approach similar to the "extend and replace" strategy. Generally, doing this requires some amount of incremental integration effort with the "present-forward" side of the 'bridge.'

  However, in its pure form, the future-back work—like Greenfield—does not produce revenue until the two ends "meet in the middle." Mitigation strategies are available, such as integrating a subset of the new Greenfield components with the partially-componentized legacy system. However, if the transformation is not completed, then the "future-back" work may be partially or entirely throw-away, making this a higher-risk approach than some others.

- In most cases where you apply this approach, you will end up with a "polyglot" system—that is, one implemented in multiple programming languages, and possibly with multiple supporting technologies around them, such as different types of databases. This is because the most natural and expedient way to implement a present-forward strategy is to componentize existing code, not rewrite it. This can leave your system in a state that is more challenging to maintain, with higher support costs, than one constructed "from the ground up" entirely with new technologies.

- While the integration effort (and data and other migration costs) is significantly reduced in this particular Side-by-Side approach, there can still be a significant test effort required to establish that the next-generation (future-back) system behaves identically with the current-generation (present-forward) system for the components it will replace. This means that the integration and migration effort is still non-zero.

It's not our intent to be unduly negative about this or any other migration approach. All the approaches offer unique benefits that can make them the best-suited option for your unique situation. However, no approach is a panacea. Our goal is that you make your choice with open eyes, understanding from the outset that there will be challenges as well as upsides. The silver bullet is picking the approach that best fits your situation, and where the downside and risks are worth taking for you.

Side-by-Side Approach
Pros and Cons

Earlier, we discussed various aspects and variants of the Side-by-Side approach to digital transformation.

While the Greenfield and Gradual Evolution strategies have their place, a variant of the Side-by-Side approach is what we most commonly use and see used in digital transformation projects. This is because this approach is flexible enough that it can almost always be tailored to the specific long- and short-term transformation priorities of your environment and organization.

Depending on your current system, a Side-by-Side approach can often be optimized to deliver (though not all at the same time) one or more: quick wins, high organizational upskilling and transformation impact, low implementation risk, or invisibility to end users. The cost is that most Side-by-Side approaches will require some degree of throw-away integration work, which consumes both time and money. Also, until you finish, they can leave your system more complex than when you started.

The Gradual Evolution Approach

Gradual Evolution Approach

Earlier, we discussed various aspects and variants of the Side-by-Side approach to digital transformation.

While the Greenfield and Gradual Evolution strategies have their place, a variant of the Side-by-Side approach is what we most commonly use and see used in digital transformation projects. This is because this approach is flexible enough that it can almost always be tailored to the specific long- and short-term transformation priorities of your environment and organization.

Depending on your current system, a Side-by-Side approach can often be optimized to deliver (though not all at the same time) one or more: quick wins, high organizational upskilling and transformation impact, low implementation risk, or invisibility to end users. The cost is that most Side-by-Side approaches will require some degree of throw-away integration work, which consumes both time and money. Also, until you finish, they can leave your system more complex than when you started.

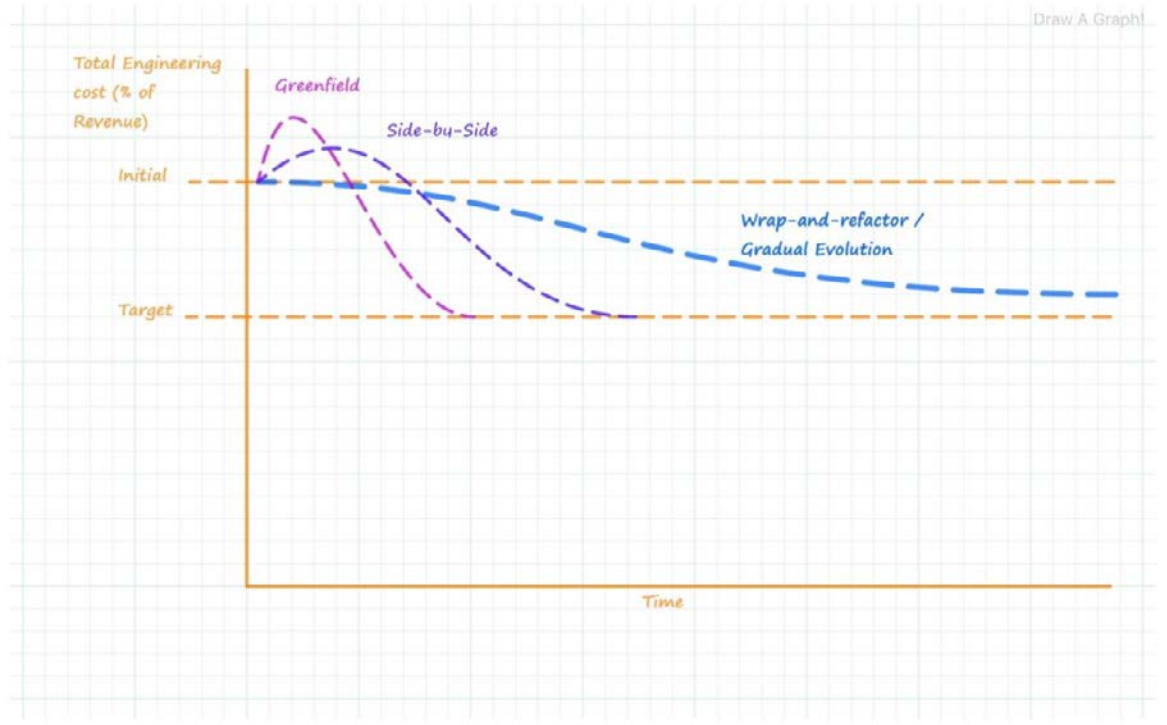

Figure 18: Gradual Evolution cost curve

So far, we have discussed the economics that drive the three major technical approach to digital transformation (above): Greenfield (above), Side-by-Side (above) and the Gradual Evolution approach.

The Gradual Evolution approach is the one people generally have in mind when they say they need to "refactor their current system". This can be an excellent way to proceed when your current architecture is a good one, and where the only real need is to manage a lightweight accumulation of quality debt. Regular refactoring is the foundation of a robust system. If your organization has kept current with the latest technology and business needs, and if your quality debt can be realistically addressed within a year for a tiny fraction of your engineering budget (for many organizations, this means single-digit person-months to single-digit person-years of effort), then a Gradual Evolution option is your obvious choice.

Where people get in trouble with the Gradual Evolution approach is when it is used to address years or decades of accumulated quality debt, or to implement major re-architecture or technology upgrades with the expectation of a quick payoff. Even where major change is required, the overall approach can be effective. In fact, we'll give a real-life example of such a (technically) successfully "evolved" large-scale project.

However, success where major change is needed means the sustained level of effort will be significant (for many organizations, tens to hundreds of person-years of effort, or more), and that the challenges faced will be big ones. In these situations, it's the mismatched expectations of "quick and easy" as the end and Gradual Evolution as the means that can lead to issues.

As with all the transformation approaches, the only 'sin' is to pick the wrong one for your actual situation. Even that is often fixable if you and your organization are honest enough with yourselves to change course when you see your current approach isn't meeting its goals. For example, if you have major quality debt and go into a Gradual Evolution transformation expecting immediate results and a low-cost, quick-win, you are very likely to be disappointed and may want to consider changing course to a Side-by-Side approach.

On the other hand, if you choose the Greenfield approach when the only real business problem could be addressed by minor refactoring, your transformation initiative will become a distraction rather than a solution. In that case, you'd be better served by abandoning your new system and refactoring your current one. As with all engineering, defining the right job to be done and then using the right tool to do it is the key to success.

So, what is the Gradual Evolution approach?

The basic idea is to start with the code base you have today and continuously change it until it's the code base and system architecture you want. In the "pure" version of this approach, there are no major subsystems thrown out and re-written—at least not until the transformation process is complete for those subsystems. The major goal here is continuity, minimum creation of new code / maximum reuse of existing code, and a gradual process of transformation and improvement.

This process is well illustrated by one approach to implementing it which we will refer to as the "wrap and refactor" approach.

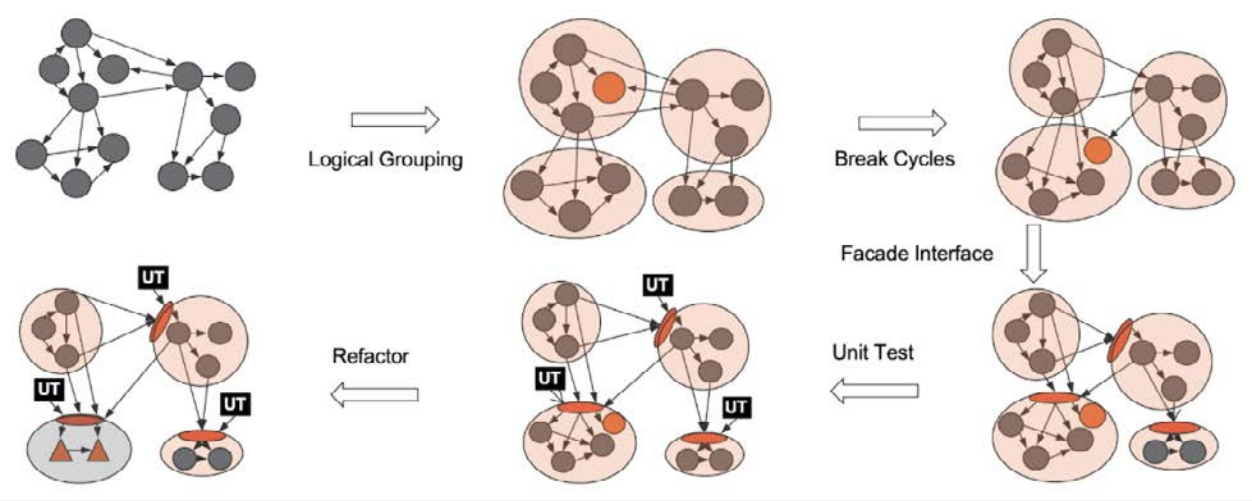

Figure 19: Gradual Evolution using the "Wrap and Refactor" approach

The “wrap and refactor approach” has several steps:

1. First, define the logical component structure you wish to achieve once your system has been refactored. This logical structure need not have any physical manifestation in the current system, though sometimes it does. For example, the software deployed on a particular server may constitute one (or more) logical groups of functionality or those that perform a specific function (report formatting, for example).

2. Next, map the logical component structure you want to the code that currently exists. Depending on your implementation, the unit of code used in this mapping might be blocks of functions, SOA service implementations, JEE components, OO class headers and implementation files, sets of scripts, or some other logical grouping. At this point, mapping is strictly “on-paper”; no code changes have occurred yet.

3. Next, refine your mapping by noting which of your logical components have cyclical dependencies in code. For example, if component A calls an element of component B, and if component B in turn also calls an element of component A, then we have a “cycle” where the two components depend on the other. A “cycle” is an inherent flaw in a component architecture since it means your components are not independent of each other; in particular, they cannot be independently replaced.

   In this example, if we replaced component A, we would potentially have to replace component B at the same time, since the two have a mutual dependency. Components with such mutual dependencies are characterized as “tightly coupled”; that is, each component depends on the detailed implementation of the other. This tight coupling is a key factor that we seek to fix in the refactoring process. When there is a cycle, the simplest way to eliminate it is to combine the two components into one; however then you end up with some very large components, defeating the goal of componentization!

   Another and better way is to identify the elements of component A on which component B depends, and to move those elements from component A to a new component, component C. Now component A and component B

both depend on component C, but they no longer depend on each other; the "cycle" has been broken. Breaking all the cycles in the new logical architecture can be complex, and may require splitting existing functionality into new parts, which requires code changes.

4. Once all cycles have been broken, the architectural (structural) tight-coupling between the newly-defined components has been eliminated. Our next concern is to eliminate the dependency on the implementation of one component by the implementation of another. The way we do this is to create a "façade" or API on top of our new components, and insure that each component is only called through this façade. This is the "wrap" aspect of the "wrap and refactor" method—since we "wrap" each component with a façade. Doing this is where the major coding work happens. Reliance on shared resources (such as databases) and direct calls to internal elements of, say, component A by component B are likely to be significant aspects of any current-state architecture with significant quality debt.

   Once these facades are in place, and once we insure each component is only called through its façade, we can say we truly have a componentized architecture. Once we've accomplished this, we have insured that there are (a) no circular dependencies between components, and (b) that no component relies on the implementation of another component. The only reliance will be on the interface (façade) of the component, not its implementation. Once this step is completed, the architecture can be said to be "refactored," since it now physically embodies your desired logical component structure.

5. Now that the architecture has been refactored, and the code has been componentized or "wrapped", the next step is to ensure that we can effectively "refactor" or "update" the code inside each individual component, without negatively impacting the behavior of the system overall. Note that "refactoring" in this approach is used in two senses: one for the architectural refactoring (already achieved in this step), the other for refactoring the code that implements each component. Refactoring the implementation code is now theoretically possible since each component only depends on the façade of the other components—the code that implements that façade should be irrelevant to the operation of the system.

While that's the theory, it's important to establish that this is also true in practice. An effective way to do this is to structure a set of unit tests against the façade or API of each component that thoroughly exercises it. The façade should be exercised as thoroughly by the tests at it would be by the system in operation, or even more, to allow for future evolution of the system.

The goal is that any new implementation of a component which passed the same suite of tests could seamlessly replace the current implementation. What generally happens in practice is that when a component implementation is initially replaced, additional calls from other components that do not go through the façade are discovered. This requires additional refactoring of the calling components and new tests, but a strong and complete set of component unit tests can be eventually reached.

6. Finally, we are at the stage where the implementation of each component can be freely replaced and "refactored", with a strong suite of unit tests ensuring that any changes to a given component will not negatively affect the system overall. This is the desired end state for a wrapped and refactored system. Such code refactoring may now be done as aggressively as desired to achieve performance improvements, to implement technology updates, to improve maintainability, or for many other reasons.

There are other ways to continuously evolve a given architecture into another, but where the end state is a modern, componentized system, they generally follow a similar pattern.

Gradual Evolution Approach
Case Study

We've described the "Continuous Evolution" approach to digital transformation, with "wrap and refactor" as a concrete illustration of one way to do it. Now we'll take a look at a real-life example of where the continuous evolution approach was (technically) successful in migrating a large-scale, monolithic, tightly-coupled system to a better architecture.

A new client had been in business for about ten years when we began working with them, but they had tapped into a dynamic and rapidly growing market and instantly achieved success. Though they had grown to a sizable company before we met them, their customers (B2B) were many times larger still—a classic "enterprise software" scenario where a relatively small company serves a relatively small number (ones, tens, or hundreds) of very large clients.

In this business scenario, it is very difficult for a company to say "no" to a new and large customer's requests for new features and tight schedules, especially in the face of evolving competition. While the client had tried hard to keep to a single code base and a common well-defined platform API, they ended up with a lot of customer-specific code bolted onto their core system. Much of that bolt-on code performed similar but slightly different functions, and all of it used complex and varying configuration mechanisms understood only by the people who had worked on each individual project. The need for client-demanded quick fixes and patches further compromised the platform code base, which had grown to over 10M lines of deeply entwined code.

When we began working with them on an advisory basis, the major issue voiced by the client was that the code had become so complex, and the configuration options so numerous and overlapping in side effects, that it took 6 months to stand up a new "default" instance of their system for a new client, even with no customization. In addition, on-boarding new development staff (which was entirely on-shore at that point) required at least 6 months before a new senior engineer could become productive.

Retention was also low because the code base had become so frustrating to work with, in part because of the unexpected side effects of any given code change. Project schedules were routinely late by factors of two or three times, and individual features were sometimes mis-estimated by a full order of magnitude. On top of all this, new competitive threats were beginning to emerge, with only the client's established position keeping them competitive.

On the positive side, senior individuals on the client's engineering team—though rather burned out—were truly talented. The client had started out originally with a well thought out SOA architecture, though the originally separate service implementations had merged over time to become, essentially, a single monolithic block of tightly-coupled code.

In spite of this, the client maintained a firm separation between the Web clients and the back-end platform that supported a well-defined API common to all customer deployments. The API was attached to the back-end system—now in effect a single, massive 10MLOC service—through an enterprise service bus. Their current client systems were Web clients, though mobile was an increasing requirement.

Because of the client's expertise in, and affinity, for SOA, they elected to continue using SOA technologies from the alternative end-state architectures we presented to them—which included our recommendation of a cloud-native alternative. In addition, because of their very large investment in the existing code base, both financial and (I think it's fair to say) emotional, they elected to take a continuous evolution approach.

Because the client had done a good job keeping their APIs refactored and common to all clients, and because these APIs were well "productized" (documented, readable, and largely - though not entirely - orthogonal to each other), the APIs themselves implied a natural component structure for the code. Because of the compromised code base, these components were no longer embodied in the legacy core system as well-defined and localized physical service implementations. Nor were they "orthogonal"—that is, they tended to interact with each other in unexpected ways. However, as a logical construct the existence of services were still very much implied.

Given the constraints of continuous evolution and SOA as an end-state architecture, we outlined a straightforward variant of the "wrap and refactor" approach for this system. The "logical grouping" of components was inspired by the system API: The logical components in the system were a cleaned-up version of the SOA services whose existence was implied by the system APIs. These services no longer had physical boundaries in the code given the compromises to the code base itself, but logically these services still had a clear identity. The system API itself, while generally good, had atrophied to some degree as well. We took the opportunity to provide a means to update and clean up these APIs in a controlled way in the course of the refactoring process.

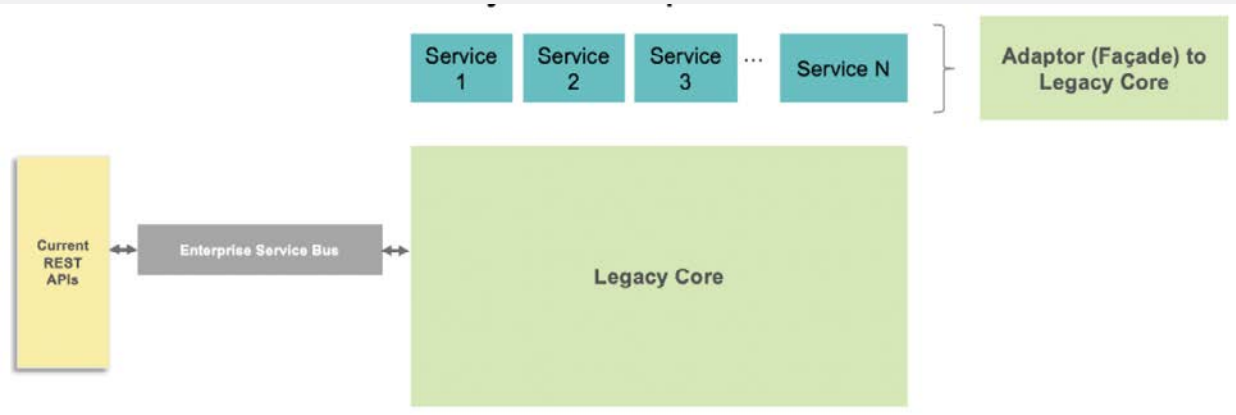

Figure 20: Gradual Evolution Example Step 1: Wrap logical services with a facade and refactor code to remove cyclical dependencies between logical components

Without altering the client-facing system API itself, the first step was to create the "as desired" service facades on top of the legacy core. The next step was to break the cycles that existed between these new, logical services through some minimum refactoring of the legacy code base.

The next step was to create unit tests for each of the new, logical services we created in Step 1. These tests ensured that the service boundaries were clearly demarcated, and also would provide a basis for ensuring the current system APIs were not broken by subsequent refactoring.

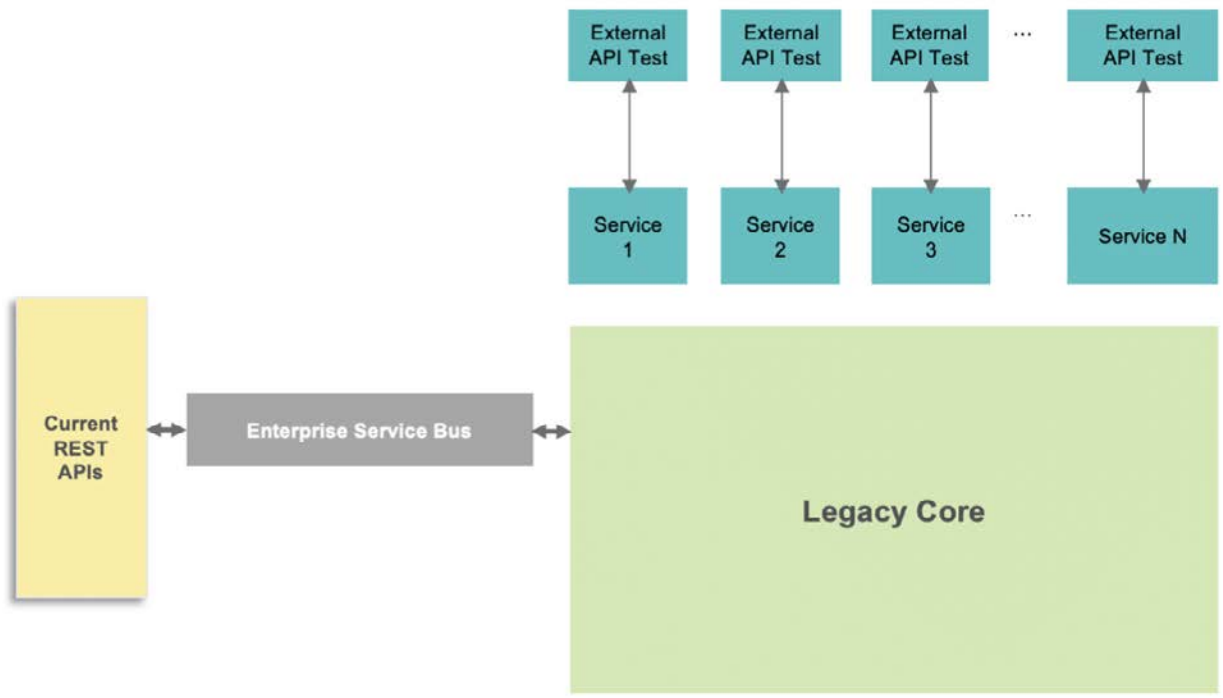

Figure 21: Gradual Evolution Example Step 2: Create unit tests to define boundaries of logical components and ensure reverse compatibility of refactored cod

Up until this point, the platform code paths exercised by the system APIs were impacted in minimal fashion, if at all, by the overlay of the new similar-but-different logical structure on the legacy core and the subsequent breaking of cyclical dependencies between them. With the unit tests in place, we would now be in a position to begin more aggressive refactoring.

The next step was to put the new logical facades "in series" with the system APIs. This meant that all calls to the legacy core now went through one or more of the new service facades, and only through those facades. Orchestration and transformation within the enterprise service bus was used to map the system APIs to the newly refactored services facades which, while similar, were not identical due to the refactoring of the services model.

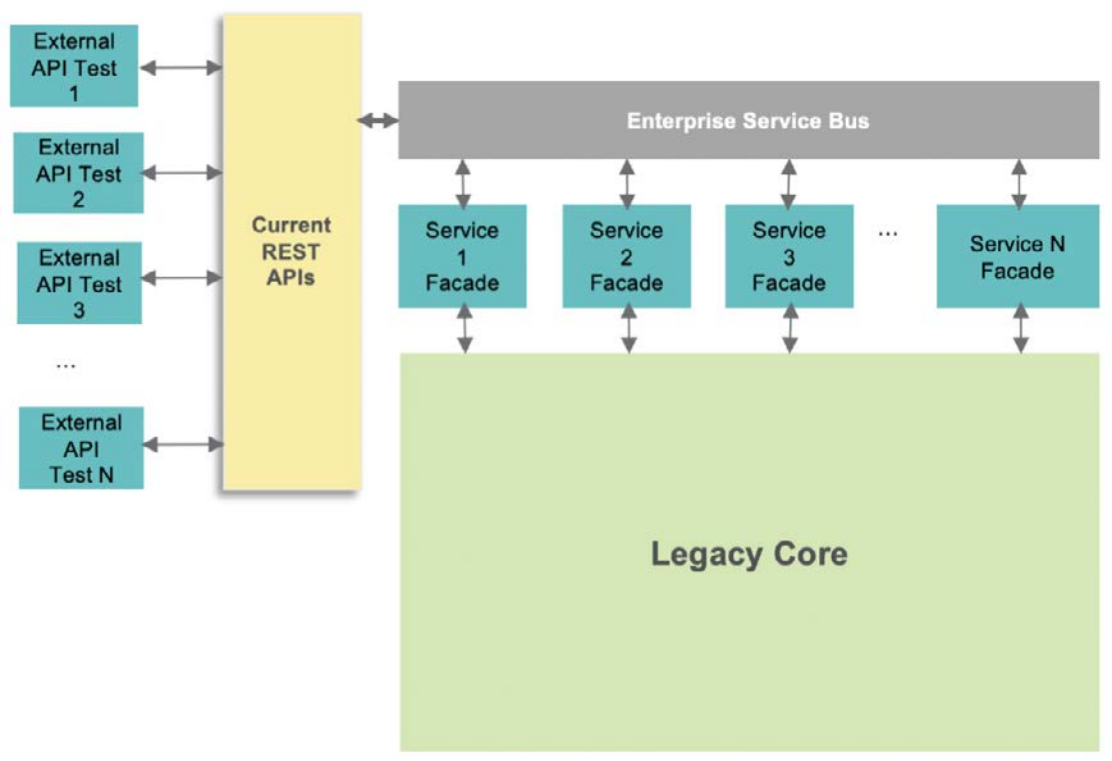

Figure 22: Gradual Evolution Example Step 3: All calls through System APIs mapped (via ESB) to new logical service facades; unit tests migrated to system API

In addition, the API unit tests were migrated to now call the back-end through the legacy system APIs. While this migration may seem wasteful, the unit tests that were configured to call the new service APIs directly might still be needed later, after the system APIs themselves had been refactored to reflect the new structure.

With both the unit tests and the service facades now in place, we were in position to aggressively refactor the monolithic legacy core into discrete service implementations without impacting the current clients. The bulk of the work lay in untangling the code of the monolithic core, as well as rationalizing and deconflicting the corresponding configuration parameters (not shown in the diagram). This could now be done logical-service-by-logical-service.

Note that in this approach, at every point the code base supported the full set of product functionality in production. In this particular case, where we are returning a compromised but formerly SOA-centric architecture to its originally envisioned, fully componentized state, there is a clear argument that at each stage the system also got less complex and easier to support and enhance as it became more modular. This lowered the cost of maintenance almost from Day 1 of the exercise.

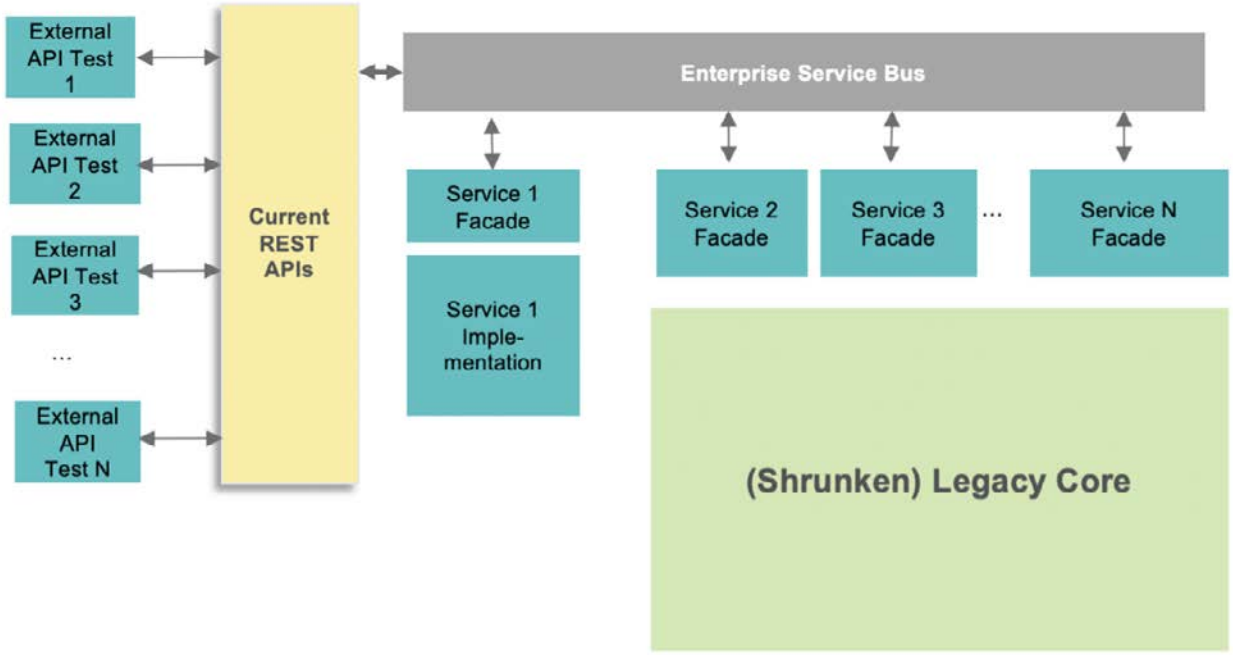

Figure 23: Gradual Evolution Example Step 4: Refactor legacy core into services front-ended by service facade, while rationalizing configuration options

Refactoring was now in a state where it could continue service-by-service. This continued until only a vestige of the original legacy core remained. The bulk of the engineering work happened in this refactoring phase.

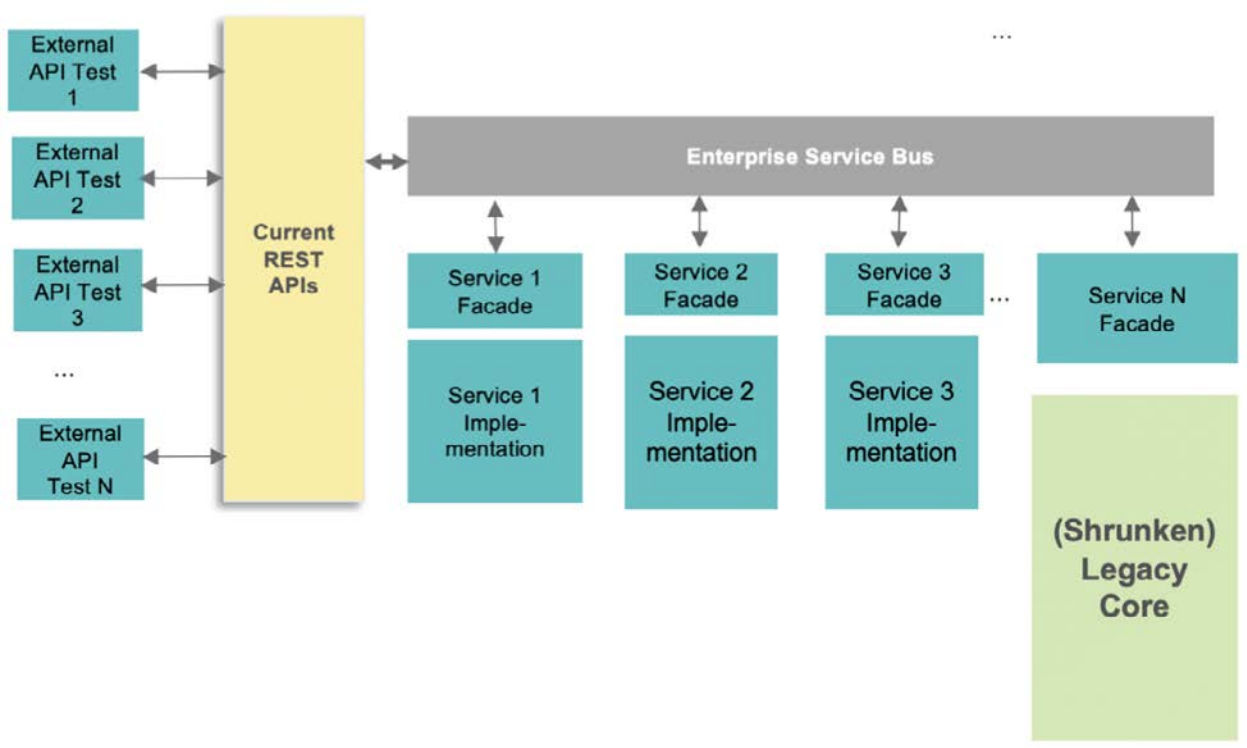

Figure 24: Gradual Evolution Example Step 5: legacy core refactored service-by-service

At this point, the legacy core was refactored to the full extent practical, with only a few functions reliant on services provided by the remaining legacy core. While outside the scope of this engagement, the system API could now be refactored independently of the back-end implementation, since we could now use the ESB to translate between the front-end API calls and the back-end service implementations.

The end result was a fully refactored legacy core with decoupled services supporting a good level of abstraction between the system APIs and the core services. The system was in good shape to support rapid evolution with a high degree of maintainability.

This is a real-life example, though, so the lessons to be learned from it are not unmixed. Let's look at a little more of the background to help us frame those lessons.

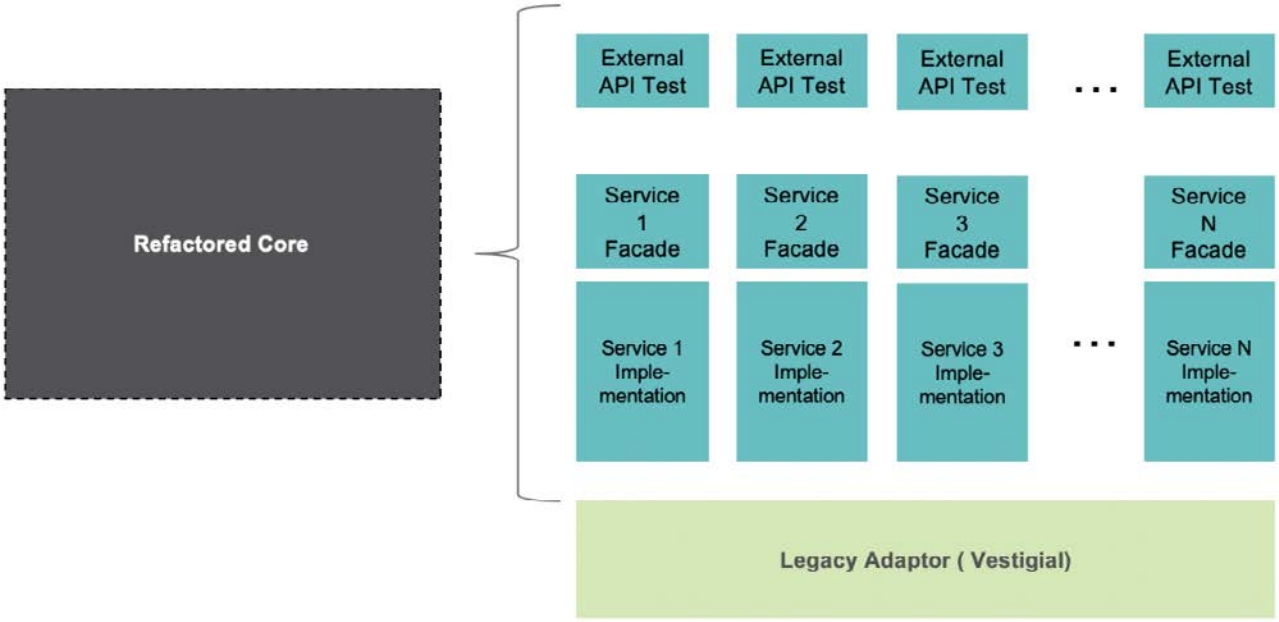

Figure 25: Gradual Evolution Example Step 6: Fully refactored core

GlobalLogic's role in this particular transformation engagement was "advisory only." Specifically, we were engaged to assist the client in architecting options for their end-state next-generation solution, and to help them visualize and plan alternative migration strategies. This client was highly conservative, both in their chosen end-state technology stack and in their transformation approach. Given the competitive situation, GlobalLogic recommended adopting either a fast Side-by-Side or Greenfield approach, culminating in a next-generation "cloud-native" end state.

However this advice was rejected by the client. Both the recommended end state and the recommended migration options were judged by the client to be too risky, and not aligned with current skillsets for the bulk of their team. Instead, the client selected the SOA-centric Gradual Evolution migration alternative—the approach outlined above—rather than the recommended alternatives.

The benefits, as perceived by the client, were that the Gradual Evolution path delivered continuous progress, and that the SOA / SQL technologies were familiar to the client's engineering staff. Overall, the client saw the approach they selected and executed above as the "safe" choice.

Following our advisory engagement, the client executed the Gradual Evolution strategy using a mixture of their own internal resources and a pre-existing partner. From a technical standpoint, the Gradual Evolution journey described here was indeed successful. In terms of cost, as expected, it took several calendar-years and several hundreds of person-years of engineering effort to execute the transformation.

To put this cost into perspective, while lines-of-code has (very legitimately) lost its value as a primary productivity metric, let's use the fairly arbitrary figure of 1,000 lines of production code per developer-month as indicative and see where it takes us. Given the client's 10M lines-of-code code base, a full re-implementation of all 10 million lines of the client's system would have taken roughly 10,000 person-months of effort (say, about 850 person-years) according to this metric.

Refactoring the entire system for several hundred person-years (i.e., single-digit thousands of person-months) of effort is therefore, on the surface, a tremendous bargain—being only a fraction of the total implementation cost. This type of thinking—that it must be cheaper to start with what you have today—is often a key motivator for those who adopt a Gradual Evolution approach, explicitly stated or not.

However, there are some fallacies to this reasoning:

- Using modern technologies and a fresh architectural approach, it is highly unlikely that all 10M lines of code would be required to replace the full scope of the client's system. Though the Greenfield level of effort was not scoped (because the client chose not to take on the perceived risk of a re-implementation) it is likely, on point examination, that a custom code base roughly 25% the size of the legacy organically-grown system would have been required for a full re-implementation, provided the new system leveraged current-generation open source components.

This makes the level of effort of a next-generation Greenfield approach comparable to the effort expended in the Gradual Evolution approach—or at least commensurate. A Side-by-Side approach could, of course, have been begun for much less, depending on the functionality to be deployed in the new system.

- Except for new installs and customer-mandated functionality, in the course of executing the Gradual Evolution approach other planned roadmap features, development was largely deferred during the course of the migration in order to fund, staff, and manage the refactoring initiative. This meant that the client's feature set stayed largely static during the multi-year course of the migration, while the client's energies were directed on evolution.

Technical success aside, during the several years it took the client to execute this Gradual Evolution approach, and with their feature set largely static, the client's share price and valuation declined by more than 90% from the value it had at the beginning of the initiative (this is a real-life example). A company widely regarded as a "scavenger" in its industry subsequently acquired their assets.

As in all such situations, this decline clearly must have been due to many factors, internal and external. In particular, it would be wrong to attribute this decline in valuation solely to the client's choice to use a Gradual Evolution transformation approach. However, taking several years of time and engineering focus to deliver a system that does basically the same thing as it did at the outset clearly must have played a role—especially during a time when the market was changing rapidly and the company was under competitive threat.

This real-life example is almost a parable of the main point of this paper: the most important factor in the overall success of your digital transformation is not so much how well you execute it, as it is choosing the *right approach* to begin with. The right approach is the one that aligns with the true "economics" of your situation: how much time you have, internally and externally; what resources can truly be focused on it without distracting you unduly from staying competitive; and how sustained your will and the will of your company and its investors are to see it through.

To paraphrase mid 20th-century management guru Peter Drucker, it's better to imperfectly implement the right strategy than to flawlessly execute the wrong one. You can almost always recover from mistakes you make in the course of executing the right strategy; but executing the wrong strategy, however well you do it, can still be fatal.

In this particular case, whether it was the primary factor in the company's decline or not, the approach selected clearly did not align with the external situation of the company. It was clearly a good fit for the *internal* dynamics of the company; that is apparent from the fact that the execution was technically successful and—in terms of a Gradual Evolution approach —relatively quick. It took "only" a few years, as opposed to decades in some companies (it was helpful here that the 10MLOC code base was only medium-sized by enterprise standards).

What was clearly underestimated by the client, however, was the *external* impact of a several-year hiatus in aggressive feature roadmap development. This particular client would have been better served by an approach that could deliver tangible business results in months rather than years, like the Side-by-Side approach. This client's choice of what seemed to be a low-risk strategy from an internal perspective turned out to be not only risky but suicidal from an external perspective.

Please do not take the wrong lesson from this concerning the Gradual Evolution approach. Even though it turned out to be the wrong choice in this situation, the financial failure of this particular company is not a sign that the gradual evolutio" approach itself is inherently bad or always the wrong choice. It was simply the wrong choice for this company given their economics.

In other cases, Gradual Evolution can and does work well. If you are in a situation where there can be an immediate *positive* financial impact from simply refactoring your code—for example, if it enables you to migrate from an expensive hosting option to a significantly less expensive one—then Gradual Evolution may be a good strategy (or a part of your strategy).

In other cases, especially for small- to medium-scale systems, Gradual Evolution can be the fastest and most effortless path to make your system supportable, putting you in a position where you can compete and grow. Also, in situations where you have lost expertise in the implementation of an existing system—such as sometimes happens when you receive the code base through a second-hand acquisition, or through attrition of key staff—then Gradual Evolution can be your quickest path to a maintainable system and full ownership. However, when using this or any other approach, one should always be mindful of the economics. In particular, pay attention to the full cost in time, effort, and in the deceleration of your forward-looking feature roadmap.

Gradual Evolution
Pros and Cons

Positives of a Gradual Evolution approach include:

- Does not immediately require new skillsets
- High transformative value for the existing team over time
- Odds are quite good there are members of your team who only need your support to get started

Some downsides and limitations of a Gradual Evolution approach include:

- "You can't always get there from here." The constraint to move continuously from what you have to what you want poses some strong limitations. In the industry, we like to say, "Of course it's possible—it's software!" And of course, at a certain level this is true. It's hard to imagine two architectures, however different, that in theory could not be morphed continuously from one into the other.

  The challenge, however, is posed by the constraint to make the evolution between the two states continuous; that is, through gradual modification of your then-current system. This constraint can make the migration far more expensive and time-consuming than allowing a discontinuity, such as the "NextGen" component of one of the Side-by-Side variants. In other words, to get the perceived low risk of continuous (as opposed to disruptive) evolution, you may need to expend considerable time and money.

- In high school mathematics, you probably learned the "intermediate value theorem." This theorem states that if a continuous function has two values, then it must at some point assume every value that lies between those two values. For example, if the value of a continuous function is 9 at one point and 17 at another point, then that function must also assume every value between 9 and 17. While there may not be a proof that this applies to complex software systems, in practice it's intuitively correct.

  In more advanced mathematics, we learn about the continuous deformation of solid objects. It turns out that in topology, you cannot continuously deform

some types of solid objects into other types of solid objects simply by stretching and bending them, even if the object is made of very stretchy material. For example, no matter how you push and pull, you cannot transform a stretchy doughnut into a sphere, or vice versa. This is because there is no continuous way to get rid of the hole in the donut or to put a hole into a sphere. You have to discontinuously sew up the doughnut hole or discontinuously tear open the sphere before one can become the other.

Similarly, there seems to be no way to continuously evolve one piece of software to an arbitrary end state. To get from "Point A" to your desired "Point B," you must introduce a discontinuity — or "breaking change" — which is where the Greenfield and Side-by-Side methods come in. In a Gradual Evolution approach, the current structure of the software always constrains where you can end up. To get from "Point A" to "Point B" in a Gradual Evolution approach, you have to go through every possible intermediate state of the system that lies between these two points. Depending on the system, this can be a long and tortuous path.

- In practice, what this means is that, for a given starting architecture, there are some end-state architectures that are not practical to achieve in a continuous fashion. While in an abstract sense doing so may be theoretically possible, the time and money it would take to continuously evolve from one state to the other is many times more costly than other ways to reach the same end-state.

- Many gradual transformation projects are never finished. This is because they can take long enough that the political will, the business situation, or even the management team who originally launched the initiative can evaporate before the transition is completed. In the introduction, I mentioned a company whose transformation to SOA technologies was still only partially completed over a decade later. They could persist for that long only because the company is very large, and the industry they are in is relatively (though not completely) mature. In many other cases, though, the market moves fast enough that taking several years to refactor would be seen as far too long— and the project would lose support and be abandoned for another approach prior to completion.

- While it's theoretically possible to both actively enhance an existing code base at the same time that it is being deeply refactored (e.g., by only enhancing already refactored components), in practice it is so complex to manage simultaneous refactoring and enhancement that it rarely happens. Instead, either the feature work is deferred in favor of the refactoring or, more commonly, the refactoring work is deferred to satisfy urgent customer or market demand for new features. This is another reason why gradual transformation projects are often never completed; they keep being deferred (or resources siphoned off) to meet needs that are perceived as more urgent until the will to finish them simply evaporates.

- Even without feature additions, a "wrap and refactoring" initiative against an existing production code base is often compared to "changing the wings of the plane while it's flying." If you add the need to simultaneously enhance the underlying system at the same time you are deeply refactoring it, the management complexity of the resulting effort would perhaps be more analogous to not only changing the wings on the flying plane, but also morphing the fuselage from that of a Boeing 737 to an Airbus A380. While somewhat hyperbolic, the challenges of accomplishing two unrelated goals in the same code base while it is being deployed has defeated many organizations, resulting in one or the other goal never being met.

- A more radical Greenfield or Side-by-Side approach has the potential to deliver new business value faster.

Pros	Cons
It has the potential to gradually reduce engineering costs and technical debt from the outset,without the initial upfront investment required for a Greenfield or Side-by-Side approach. This can make it the only politically viable approach in some situations.	Many systems undergoing gradual evolution remain more-or-less at feature parity throughout the process. Change is not necessarily visible, either to customers or to internal stakeholders. This can be good or bad, depending on the situation.
No new technical skillsets are required to implement this approach. The current team can start from where they are today and immediately make progress.	For a complex system, implementation of this approach can take several years or longer. Because of the timeframe and because results are not necessarily visible, the political will to carry the transformation through to completion can vanish. The system will then be left in a partially converted state that is more complex to maintain than it was initially.
It has the highest tranformative value for the current team, as they work on both the “old” and the “new” systems.	Because there is generally a single team with common skillsets, the best resources working on evolving the current system tend to be syphoned off to deal with urgent problems, new sales opportunities, and business-as-usual. This means the transformation effort is either slowed or becomes start-and-stop.
	The requirement to continuously evolve from the current state puts limits on where you can end up. To get to an arbitrary (and desired) end state, you generally need to introduce a discontinuity. In other words, this approach in its pure form can only get you so far (though in a given situation, that may be far enough).

The ROI of Digital Transformation

The ROI of Digital Transformation

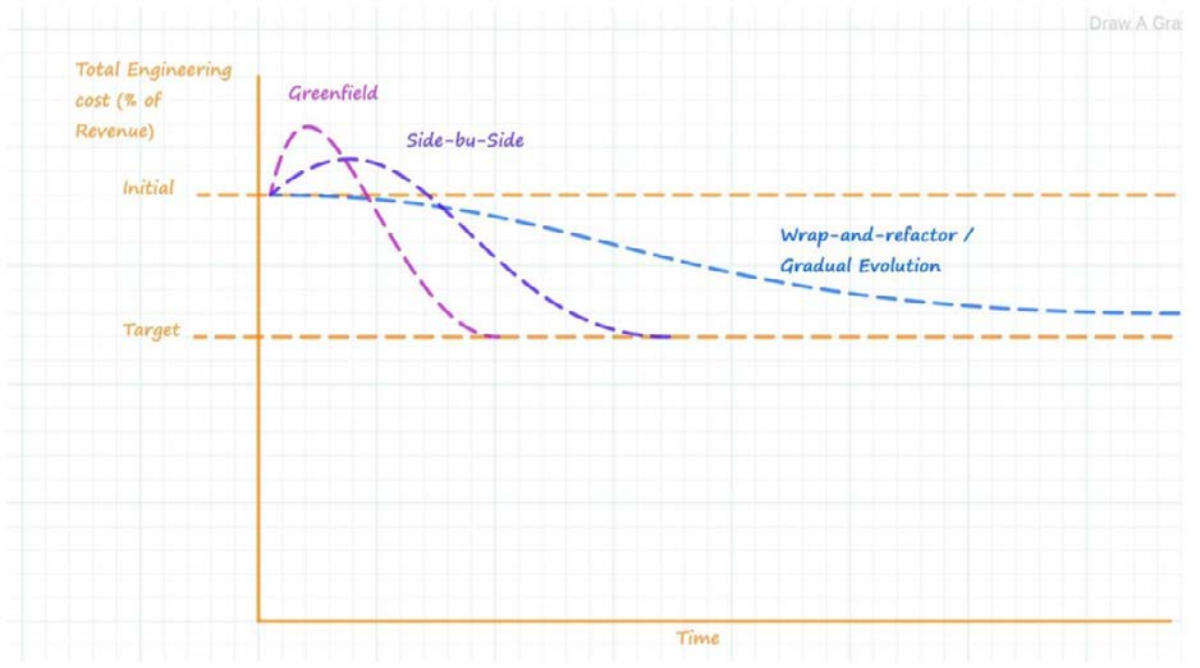

Figure 26: The ROI of digital transformation approaches

In this paper, we have characterized the functional "economics" of digital transformation in its broad textbook sense: deciding how to wisely deploy the finite assets you now possess in a context of constraints and opportunities to achieve a goal in the future. Our major focus thus far has been on the "decision" aspect of how to proceed. Now let's discuss the Return on Investment (ROI) of a digital transformation initiative.

The reason we deferred the ROI discussion until now was so we could first make the case that digital transformation, when it's necessary, is about nothing more nor less than the continued viability of the enterprise. The ROI of a transformation initiative is a discussion of how much the prosperity or even the future existence of a company (or line of business within a company) is worth. If we had made such a strong statement earlier, it probably would have seemed like we were overstating the case. However, equating transformation with viability is nothing more than a simple statement of fact; there is nothing hyperbolic about it.

Digital is the current imperative behind nearly all business interactions, and many personal ones as well. The need to modernize to stay competitive is nothing new in human history. We know that any organization that fails to respond to the current demands of both its internal and external environment tends to become "food or fertilizer" for those that do adapt. It's harsh, but it's the "economic" and "environmental" reality that humans and human endeavors have always lived in.

This means that any ROI calculation or investment decision must take in to consideration that digital transformation is an existential necessity for organizations. Transformation is not so much about developing incremental sources of revenue as it is about rethinking who you are and what you do, given a new environment. A focus strictly on incremental revenue may imply a mindset that can lead directly to the failure of the enterprise.

We will anchor this financial discussion around two well-known companies who both launched technology transformation initiatives back in the 1990s, one successful and one not successful. While Apple's technology transformation to its OS X operating system starting in (fiscal) 1997 was not in itself inherently "digital," it enabled Apple's later success and dominance in the digital realm. Kodak, on the other hand, explored new technology in the 1990s and before that was indeed literally "digital."

There was a serious attempt within the Kodak company to move from a total reliance on physical goods, such as chemically processed film and prints, into the then-new realm of digital photography. Kodak began investing in digital in time to begin offering products in the early 1990s, with their 1995 annual report describing the recent release of digital products which "capture, store and print images in an electronic format."

The reason for choosing these two companies is that they are both iconic, similar in size, and public (i.e., financial information is available and not confidential). Both companies had an initial reliance on physical products (i.e., computers/printers/hardware vs. cameras/film/chemicals), though Apple had the advantage of a significant software ("digital") component as well. Both companies saw the future in their industry and responded by launching major technology initiatives, with a comparable dollar investment relative to the revenue of each company. And since both companies launched their initiatives in the 1990s, we have two decades worth of data to decide how things truly worked out for Apple and Kodak.

Transformation Approaches

Using the transformation terminology we've defined so far, both Apple and Kodak followed a predominantly Greenfield strategy, with elements of a Side-by-Side approach mixed in, to enable their legacy customers. Apple purchased the NeXT operating system and used it to retire and ultimately replace their in-house Mac OS operating system—a "Greenfield through acquisition" approach. Kodak developed new types of cameras and related storage and other infrastructure for digital, including online photo sharing systems and in-store printing kiosks—a "Greenfield through build" approach (although there were also acquisitions).

Apple maintained a "classic" environment Side-by-Side within its new OS X operating system. This allowed legacy Mac OS users to continue running their existing Mac OS 9 applications inside a "shell" supported by the new operating system. However, Apple's explicitly stated goal from the beginning was for current Mac OS customers to transition from the old to the new. Apple even staged a mock funeral for legacy OS 9 at one event.

Apple stopped actively enhancing its legacy Mac OS 9 operating system in 2001, the same year that the new "transformational" OS X operating system shipped. Apple continued to provide dwindling levels of backward compatibility in OS X for about 8 years (without making enhancements to their legacy Mac OS) until backward compatibility was removed entirely around 2009. Apple's Side-by-Side approach essentially provided a one-way ticket from legacy to NextGen, whether their loyal legacy users wanted to go there or not.

Kodak, by contrast, focused on the continuing co-existence and interoperability of physical and digital photography. Kodak's online digital photo sharing sites and in-store kiosks were oriented around allowing users to make physical prints of digital photos, as opposed to taking a "digital only" or "digital native" approach. Kodak focused on both directions of the physical and digital divide, supporting seamless migration from digital photos to physical prints, and from physical media to digital.

In addition to digital cameras, Kodak provided film scanning services and physical storage for digital photos like "PhotoCD." While it's hard or impossible to read the mind of a company, it does not appear that Kodak's side-by-sid" strategy of the 1990s and 2000s was aimed at dragging reluctant users into a digital-only world. Instead, it seemed focused on allowing easy portability back and forth.

ROI and Financial Impact

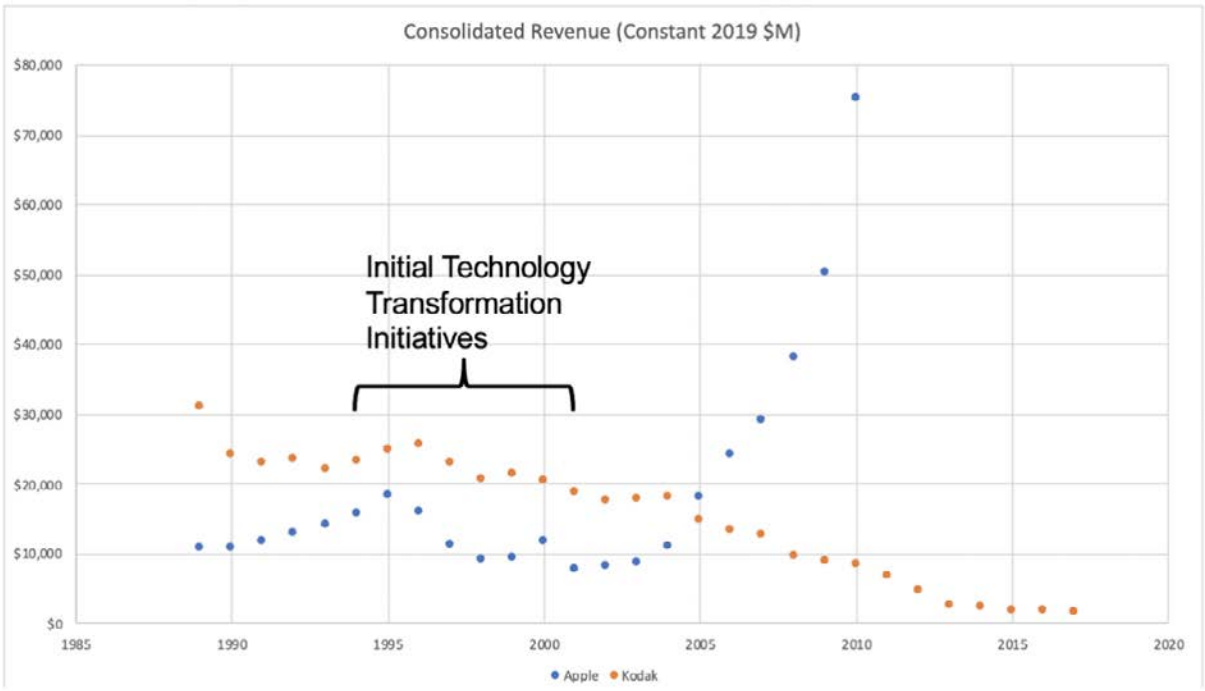

Figure 27: Initial technology transformation initiatives

We've discussed the background of the two transformation approaches. Next, let's look at the financial aspects of the two technology initiatives:

- Apple acquired NeXT Software in Apple's fiscal year 1997 for $429M in 1997 dollars ($674M in 2019 dollars). This purchase represented 6% of Apple's revenue in FY97. While Apple made and continues to make significant investment in R&D, Apple's acquisition of NeXT was, arguably, the seminal event that led most directly to Apple's future success. This acquisition brought Steve Jobs back to Apple, as well as many other figures behind Apple's subsequent success, together with the foundational technology for OS X, iOS and other innovations.

  Essentially, this was Apple's initial investment in a greenfield system that

would ultimately replace their then-legacy system, Mac OS 9. The NPV in FY1998 of Apple's future net income, using an annual discount rate of 5.5% (average Dow Jones index return over that time), and calculating through Apple FY2018 using 2019 constant dollars, is $167.4B USD. The ROI on Apple's investment in NeXT was therefore 248x (24,800%) in constant dollars. This is a dramatic over-simplification of a complex picture, but is one data point to consider.

- While the exact investments Kodak made specifically in digital do not appear to be public, Kodak clearly made early and significant investments in digital photography. These amounted to over $2B ($3.4B in 2019 USD) in the 1980s, 1990s, and early 2000s according to newspaper accounts and business school reviews. Kodak was an early entrant in the digital photography space and initially claimed significant market share in digital cameras and online photo sharing. Kodak also made a significant play to scan, digitize, and store film-originated photos onto digital media ("PhotoCD") as early as 1990.

  A total investment figure of $2B would represent 13% of Kodak's revenue for the single year 1995, when major digital product launches were called out in their 10-K filing for that year. If this investment was spread out over several years (as it certainly was), then Kodak's investment in digital in any given year would be the same or less than the 6% of revenue Apple paid for the acquisition of NeXT. The NPV in 1996 of Kodak's future net earnings (using an annual discount rate of 5.5% and this time calculating through 2017) is $6.76B USD—about 25x less than Apple's NPV at the time of their transformation. Using $3.4B (in 2019 dollars) as the amount of Kodak's digital transformation investment as of 1995 gives an ROI for Kodak of 0.99 (99%). In other words, they doubled their money.

Assuming that Apple represents a best-case scenario and Kodak a (near) worst-case, the over-simplistic answer about the ROI of a digital transformation project is that it can range from about 100% to about 25,000%.

Of course, both of these ROI calculations are wrong because they assume that the entire future net income of each company is the result of its one-time investment in digital transformation. This is clearly not the case for Apple, Kodak, or any other company. Both companies made continuing investments in R&D over many years — both before and after the one-time investment used in our

calculation. There were also many other factors besides these one-time events that were at play—not least of which was the experienced, hard-working, and talented people they had on-board and the enormous brand equity, prestige, and goodwill that had been accrued by each of these iconic brands over a long period of time. It is clearly true that even the right transformation strategy that is successfully executed is not by itself the one and only factor driving long-term success.

However, an *unsuccessful* transformation, uncorrected, is indeed enough to guarantee failure. In that sense, the entire future of a company and its future profitability does indeed depend on this one investment. Despite everything else working in its favor, Kodak's market cap fell from over $30B in 1997 to $145M in 2012—a 99% reduction in equity value over 15 years.

We saw the same thing happen in the case study we presented to illustrate the Gradual Evolution approach, where the value of that company also declined by over 90% in just a few years while they efficiently executed the wrong strategy for their situation. In the Kodak case, the enormous brand equity, size, and depth of talent of that company bought them a long, slow decline, while for the near-startup in the Gradual Evolution case study, the decline was just a few years. But in both cases, we see an unsuccessful transformation attempt leading to the literal failure of the business.

Another contrast between Kodak and Apple is that while Kodak kept its primary focus on photography, Apple broadened its scope to other offerings, such as digital consumer electronic devices and associated consumer digital services. Apple even went so far as to drop the word "computer" from its name in 2007 when it launched the iPhone. Apple also actively developed purely digital sources of revenue, including iTunes music and movies, the "App Store," iCloud, Apple Pay, and other services.

Apple's service revenue alone in fiscal 2018 was $37.2B. While still a small part of Apple's total revenue mix (about 14%), that is only because Apple's total revenue is so enormous ($266B in FY2018). Apple's FY18 service revenue of nearly $40B is enough to comprise a large company in itself—larger than Kodak in constant dollars, even at its height.

The key takeaway from our ROI discussion is that transformed technology positions your company to stay competitive and take advantage of current and future opportunities. It's up to the company to exploit their new technology to capture the opportunities that the future presents. Apple successfully leveraged its technical success to become a "digital native," while other technically successful innovators like Kodak did not. Although Kodak was an early technology leader in digital photography, it never really transformed to become a digital native. That's why it never realized Apple-like financial benefits.

From an ROI calculation perspective, it's not possible to consider an investment in transformation as a one-time event. While the technology investment is a necessary component of a transformation initiative, it is not the sole cause of its ultimate success or failure. We will discuss other factors leading to post-investment success in more detail in the next section.

Analysis

So why did Apple succeed so spectacularly while Kodak went into bankruptcy and, from a pure business standpoint, failed? We might argue that being in the film and camera industry doomed Kodak, while Apple was fortunate enough to find itself in consumer electronics, a subsequent high-growth sector. We could say that Apple got lucky because they were in the right place at the right time. However, we don't think this is the right answer.

Kodak's failure and Apple's success are topics that many smart people have debated and will continue to debate for many years to come. And there are certainly many facets to the truth. With no claim to offering the definitive answer, we would like to present a perspective that we think will be helpful and practical in terms of your own company's potential digital transformation. Specifically, we think this is the most useful and actionable lesson for a company that now finds itself with a need to transform, as Kodak and Apple both needed to do in the 1990s.

The defining characteristic of any new technology is that it changes what is possible. Most technology changes are minor and incremental — performing a given task faster, improving quality / lowering costs somewhat, or achieving larger scale / throughput. Other technologies are transformational: they not only change how we do things, but they fundamentally change what can be done. It is not always obvious which is which. But treating a transformational technology as incremental is a certain path to failure — unless rapidly addressed and corrected — because others will realize and exploit the opportunity it presents.

With the terrific benefit of hindsight — which of course the staff of 1990s Kodak did not have (nor do we in our own situations) — what could Kodak have done differently? In addition to hindsight, we also now know for sure what Kodak may not have fully realized at the time: that "digital" is a transformational technology, not an incremental one. With that insight, and with the benefit of hindsight, what is the most important thing Kodak could have done to change the outcome?

In our opinion, the key thing Kodak could have done differently would be to take a step back and ask what business tit was really in, and if digital offered a better path to achieve it.

Even in the face of digital, a striking fact is that Kodak kept its focus on photography. One might think this is obviously the right move for them, since they were, after all, founded as a photography company. Indeed, companies like Shutterfly, Instagram, and many others have found success in the digital world with a primary focus on photographs. However, in the case of Kodak, their continued focus on photography mixes up the means with Kodak's stated end goal. Kodak could have become a true digital native and reaped Apple-type rewards had it instead kept the focus on its stated mission.

Through its own marketing, Kodak introduced the notion that a photo is a means of preserving memories and moments. The phrase "Kodak Moment" has entered the dictionary, literally, as describing a "memorable event." The Oxford English Dictionary quotes a use of that term as being in print in 1994; the phrase may be older but has certainly been with us since at least the early 1990s. Kodak's advertising slogan, publicly introduced in 2001, was "Share Moments. Share Life."

In other words, Kodak did indeed realize it was in the memory and moment sharing business, not the photography business—at least in its advertising. What it apparently did not do, and what cost it so dearly, was to seriously entertain the question, "With digital technology, is there a better way than photographs alone to preserve and share moments and memories with loved ones and friends?" In other words, is there a better means to achieve the same goal that is now enabled by this new technology?

This type of question is a very hard question for anyone to answer or even ask, because it's a very human impulse to confuse the means with the ends. There is the classic business school example of the successful "buggy whip" manufacturer in the late 1800's who subsequently went out of business because the automobile replaced the horse, destroying the mass market for horse whips. While this example has become a bit of a cliché, it's still a very effective lens from which to understand technology transformation because it is so removed from our current day and age as to be almost funny. In other words, we have no emotional connection to it. That very remoteness lets us make some very dispassionate observations.

The lesson commonly drawn from this parable is that the "buggy whip" company may have survived and prospered if it had reframed its mission in terms of

being in the transportation business; that is, if it had been clear that being a "manufacturer of buggy whips" was the means, not the mission. If our buggy whip manufacturer's mission had been expressed as "making transportation go faster,", it could have begun shipping products aligned with the latest technology shift—perhaps accelerator pedals for motorcars, or leather helmets for early motorists to wear as they sped along. If its mission had become "leather goods for the transportation industry," it could have pivoted to making leather seats and trim for car interiors.

On the other hand, if it decided to generalize on its current means of production, it could have gone down a different path, with a pivot away from the transportation industry entirely. For example, a clarified mission of "hand-braided leather goods" could have led it into making men's and women's braided leather belts. That may indeed have kept the company viable, if somewhat niche.

However, if it had chosen to keep its focus on a generalized *means of production* (braided leather goods) rather than the end (its value to the consumer through its role in transportation), it would have missed the revolution in the transportation industry that was transforming its market. Clarifying its value to the end user and figuring out how to deliver that value even in the face of the market transformation would have positioned it to participate in the rewards of the motor car revolution. This is the lesson for us as we face our own transformative technologies: focus on value delivery as the ends; don't focus on the means.

In a 2010 article in the *New York Times*, Randall Stross challenged the classic business school example of buggy whips. Without debunking the buggy whip example, he instead points out that the "wagon and carriage" companies that were actually most successful were those whose technologies already aligned with the needs of the emergent auto industry. Specifically, he cites the Timken Company, who originally made roller bearings for wagon wheels and is still in business today; the lantern manufacturers who pivoted to making headlights; and Studebaker, who originally made the metal and wood components of horse-drawn carriages but prospered in the automotive industry for decades thereafter.

These companies were able to leverage their current means of production in the new horseless world through a rapid pivot that didn't change the essence of what the company did. Stross also mentions that one of the then forty US-based buggy whip manufacturers actually did survive by becoming a niche player serving equestrians. Per Stross, few of the other 13,000 companies that were in the wagon and carriage industry in the late 1800s survived the transition.

While it's great to be in an industry whose business model happens to remain relevant in the face of a major technology transformation — such as horse-to-automobile or traditional-to-digital — these odds are not encouraging. The survival rates from 0.1% to 2.5% implied in Stross' article are not ones we would wish on any company in a period of transformation. Depending on luck is not a strategy.

Our belief is that separating the mission from the means is critical to success when a transformational technology is introduced. If you want to survive and prosper, remaining a buggy whip manufacturer is no longer a viable mission when your underlying market transforms. And even if your means largely stays relevant through one transformation (i.e., a roller bearing manufacturer for wheeled transportation), asking yourself the hard question about the value you deliver to your customers is how you continue to stay relevant in the face of future market transformations.

While roller bearings may have remained relevant for things with wheels, a roller bearing manufacturer still needs to define its value to consumers in its own disrupted segments (e.g., industrial processes where air flotation has become important). As a company facing transformative technologies, it's never wrong to clarify the true value you deliver to your customers, even if there's an outside chance that the answer in a given instance may be "Whew—we're OK!"

Kodak continued to focus on the means (i.e., photographs) rather than the ends (i.e., sharing memories) without fully appreciating the extent to which digital made other means possible. We suspect the major issue was the question of whether there was money to be made in sharing memories and moments in a purely digital age. Much of Kodak's traditional revenues came from selling physical cameras, chemical film, and processing services.

Even if the end result of enhanced memory sharing was achievable digitally, it's not good business if you can't make money from it. If we had been flies on the wall of the Kodak boardroom and executive staff meetings, it's a safe bet that this is the key argument we would have heard against Kodak's committing to a purely digital strategy in the 1990s and early 2000s.

Were the naysayers right? We have sufficient time and distance now to answer this question. If there was indeed a solid digital business in the moments and memories sharing space, then surely someone would have capitalized on it by now if there was money to be made. Who, in the digital age, has achieved Kodak's stated mission of sharing memories and moments—including digital images of those moments—as a primary business activity?

There are actually a number of what we could call "moment and memory sharing players" today. In fact, Facebook had revenues of $55.8B in 2018. Even adjusting for inflation, this almost entirely digital revenue is significantly larger than Kodak's highest-ever revenue year selling cameras, film, processing chemicals and other physical goods—which was about $30B in the late 1980s, using 2019 constant dollars. So there was indeed money to be made as a pure-play digital native fulfilling Kodak's stated mission of sharing memories and moments.

Why didn't Kodak itself seize this digital opportunity and become the dominant player? There are probably many reasons, but the main one is almost certainly that letting go of the past was an insurmountable problem. From both a corporate political power dynamic and an emotional standpoint, it would have been a profound challenge for Kodak to let go of its long history as a photography company and instead embrace becoming a digital moments and memories sharing company centered around a totally new technology.

As just one example of how hard this would be, consider the revenue streams: Facebook, the actual digital native in this space today, earns roughly 90% of its very significant revenue stream from advertising. This revenue model is a far cry from Kodak's traditional revenue streams from sales of physical equipment and related services. Embracing a new revenue model would have required completely re-thinking and re-skilling the entire Kodak company from sales to manufacturing to R&D. A radical restructuring eventually happened anyway, of course, through declining revenue and bankruptcy.

However, making sweeping changes consciously, intentionally, and while the legacy organization and power dynamic are still fully intact is a challenge of a very high magnitude. There were certainly change agents inside Kodak in the 1990s; this is clear from the progress it did make. Given that Kodak did not become a digital native company, we can conclude that the organization barriers were not overcome in time to make a difference.

If Kodak had successfully re-invented itself as a purely digital entity like Facebook, would it have saved the company? Given Kodak's current dinosaur image and Facebook's relative youth, we might think that this digital native revenue stream would have come too late. But it didn't. Kodak filed for Chapter 11 bankruptcy protection in 2012 and emerged from bankruptcy in 2013. Facebook started generating revenue in 2004, achieving a billion-dollar run-rate by 2010.

If Kodak had become the digital native in the memory and moments sharing space rather than Facebook, and if it had captured the same revenue stream (big "ifs" to be sure), then digital revenue could have been generated in time to prevent Kodak's bankruptcy and put it on an Apple-like high-growth trajectory. With the unstoppable decline in Kodak's legacy chemical photography business, restructuring costs, and the investment required to launch an entirely new digital entity, there would still have been years of combined losses in the late 2000s.

However, stand-alone Facebook was not profitable from Day 1 either. With the upward trajectory from the new digital native revenue stream, investor confidence in the combined entity could and possibly would have been retained. Seeing the eagerness and high expectations with which the actual real-life Facebook IPO was greeted, it is entirely plausible that a hypothetical combined entity would have averted bankruptcy even in the face of some losses in the late 2000's.

As a hypothetical "Digital Native Kodak," we plot the combined revenues of Kodak and Facebook against the same timeline, realizing of course that these were and remain independent companies.

As we can see from the chart on the next page, while success would have been delayed relative to Apple's early-2000s decade inflection point, the hypothetical digital native version of Kodak would have followed a similar qualitative growth trajectory to Apple's. The reason for the offset growth curve of real Apple and hypothetical Kodak is that for a digital native to prosper, the underlying physical infrastructure needs to exist. Apple's iPhone did not ship until 2007, with the Android following in 2008. Real-life Facebook and hypothetical Kodak could not truly become the digital native success story we see today until those critical components were in the hands of consumers.

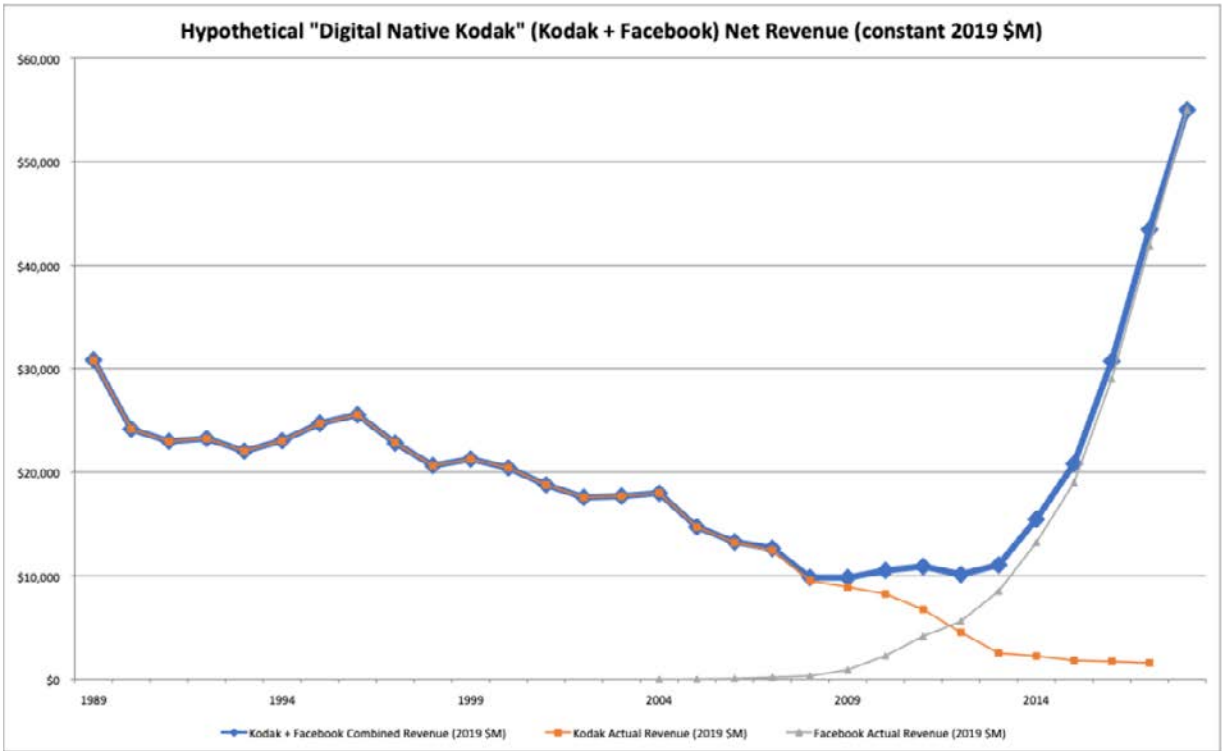

Figure 28: Hypothetical "digital native" Kodak net revenue

To be a true digital native in the memory and moments sharing space, like Facebook is now, Kodak would have had to ultimately shed its reliance on physical devices as a major revenue stream. In part because the cost of camera production got so cheap, digital native Kodak, like the real Facebook, would have needed to build pure advertising-funded service offerings around third-party cameras and other hardware. Such hardware would be supplied by third-parties like Apple and Samsung.

From Kodak's actual behavior, it was apparently not in the company's DNA to surrender the camera and print business. This failure to shed the past and fully embrace digital is the major reason why the digital native version of Kodak (and the revenue shown in the notional graph) is pure fiction. With this context, let's go back to our ROI discussion. As we implied earlier, the notion of the incremental ROI of a transformation initiative is a dangerous one because it's the wrong question to ask. Considering such an initiative as incremental can easily doom

it to failure. Certainly, one's technical migration to digital can be incremental; the Side-by-Side approach in particular lends itself to testing new business models. And the success of a given digital revenue stream can certainly be measured — the Facebook portion of our "Digital Native Kodak" is a case-in-point, as is Apple's services revenue. So, from a pure accounting standpoint, if we knew the specific investments that Apple made in its services business, we could compute the ROI of this digital entity.

However, from a management standpoint, thinking of digital or any other transformational technology as a bolt-on to your current lines of business is a very big risk. It's like using your automotive initiatives to sustain and modernize your buggy whip business. You must help digital become a self-supporting first-class citizen in your organization through the new business models that it enables, or else you have not achieved transformation. Using a transformational technology as a way to extend or supplement your existing revenue streams makes you vulnerable to those who use the new technology natively to invent new business models. You also fail to capture those potential new native revenue streams for your company.

Bottom line, in the face of a transformational technology like digital, the more you hold on to your past, and the more tentatively you embrace the new business models that the new technology enables, the closer you come to following Kodak's example. To reap the benefits of a transformational technology, you need to commit to it and the new revenue streams it allows with the full vigor of a new startup — like Facebook going after the moments and memory sharing market that *could* have belonged to Kodak. If you don't, you become vulnerable to digital natives emerging within, or moving laterally into, what could otherwise be your space.

In the next section, we will tackle the question of concrete steps you can take to end up more like Apple and less like Kodak in your own transformation process.

Why Do Digital Transformations Fail?

Why Do Digital Transformations Fail?

While far from cookie-cutter, the approaches we outline in this document are pretty deterministic. We have also made what we hope is a rather compelling case that digital is a transformational technology, not an incremental one, and that businesses fail to fully embrace it only at their peril.

Why, then, do so many digital transformation projects fail?

First, "failure" in a transformation initiative can be hard to pin down while it is happening. Companies tend to evaluate success by purely internal criteria until and unless that is no longer an option. Depending on the balance of political power in the company, this means that actual failure can be obscured for some time—by blame shifting, priority changes, direction shifts and frequent re-orgs for example. Even success against purely internal criteria may still result in failure in the real world. (See the Gradual Evolution case study for an example).

Failure to transform suddenly becomes very clear when the company hits the wall and either is no longer competitive, or literally goes out of business. Declining market share, declining stock price, and mass departures of employees eventually make it all too obvious that the transformation initiative is overdue, or has failed. It is clearly better to fix our strategy long before we start to experience these negative outcomes. But why do so many companies not get serious about transforming until they are in an actual crisis?

Three years ago we were asked to evaluate a company's forward-looking technology strategy by their new principal investor. This relatively new acquisition had seriously aging technology—known at the time of the investment—and the company's in-house team had developed a next-generation system architecture and modernization plan. Because the team and its capabilities were still largely unknown quantities to the investor, they wanted a second opinion on the forward-looking plan, architecture and technology choices, and implementation timeframe and costs. We were asked to review these factors and make our own independent assessment.

We found that the company's engineering leadership and architecture team was actually quite sharp, and their ideas about where to take their system were good ones. They had also come up with some truly ingenious strategies to extend

the life of their legacy system until it could be replaced. Together we ended up estimating that it would take about three years to completely replace and migrate the bulk (say 80%+) of their customers off of their legacy system. This time was required because the system was mission-critical to its users, highly complex, heavily customized, and feature-packed.

Fortunately, along with those clever strategies from the company's engineering leadership, we believed that the company could probably eek another three years of life out of the current legacy system without losing their customer base. Part of this retention roadmap included incremental Side-by-Side releases of new features on a next-generation platform, as well as a promise to their increasingly impatient customers of an updated system being delivered incrementally, with completion on the horizon.

Extending operations of the legacy system beyond three years would not be advisable, we concluded, because the operating costs of their legacy was very high; the customer experience and feature set was even at that time years behind customer expectations and would only get worse; and new feature implementation had nearly come to a stand-still due to the technical intricacies of working with the legacy.

We also worked with the company to help improve their next-generation architecture, and to help flesh out a Side-by-Side implementation plan and high-level roadmap that would start delivering next-generation business value promptly. Over time, the functionality of the next-generation system would allow components of the current-generation system to be retired in a phased manner, reducing but not eliminating the incremental cost of the migration. We estimated the incremental cost of the Side-by-Side migration while the company simultaneously maintained its legacy and wrote up and delivered our findings and recommendations.

The investors, the company, and we the reviewers all felt good about the proposed Side-by-Side migration from legacy to a next-generation system, and to the architecture for that next-generation system. Then nothing happened.

The decision to invest in the next-generation system was postponed and postponed. Then finally—three years after our original advisory—we were

contacted again by the investor saying that the company was now in crisis. As it turned out, they had indeed implemented the plan to extend the life of their existing system, but due to the investment required never did modernize their legacy. They had now reached a point where they could no longer compete—could we help? Of course, we will help and do what we can to turn the situation around. But at this point, the company is at best playing catch-up, and at worst in a death spiral.

This is actually the most common failure pattern for a transformation initiative: it never starts.

It took a lot of integrity for the investors to call us and admit they made the wrong call. While it may or may not be too late for this company, many successful transformation initiatives are born from the ashes of those that failed before, or those that were never started. Unbelievable as it seems today, this was the situation at Apple when it acquired NeXT and brought back Steve Jobs in the late 1990s. There was a feeling of impending doom, continuous layoffs, and despair that Apple's days were numbered. Apple decided to try one more time by acquiring a company with a new operating system, and their subsequent success is history.

Not every such story ends happily, but none do until there is a recognition of a compelling need to transform. Sometimes this takes longer than others, but recognition that there is a problem is the key ingredient for success—along with the energy, money, people, and time to make it happen.

Another reason failure is hard to identify is because it takes a long time for a significant size company to fail. In a very large company, it can take decades. This is one reason transformation initiatives are never begun. Put uncharitably, many of the people in authority are confident they can ride their company right into the ground. They believe they will have moved on or even retired before the company fails—and in the meantime, they have other priorities that bring more immediate rewards. In fact, it's much more comfortable to deny, consciously or unconsciously, that there is even a threat. Disruption is all rather theoretical until suddenly it's not. In the meantime, it can be easy to shrug off.

For people comfortable in their current position or prospects, transformation can be a threat because it changes the power dynamic. The potential downside of

transformation, for their position, is seen as potentially large while the upside is uncertain. And even if the company does fail—which is never a certainty in their mind—their current status will help them to obtain a good position somewhere else. They feel they are fine the way things are today.

Also, in a business there is always something urgent that requires top management's attention. Your board may be applying pressure to address a DSO problem, or you may need to add a couple of basis points to the EBITDA by reducing costs. Recognizing the fact that the world around you has undergone a sea-change and that your entire business strategy needs to change with it seems rather theoretical. As top executives, we can become more focused on the internal workings of our company rather than its changing role in the world around us. It often takes some type of catastrophic event or trend to shake us up and get us to take stock.

These and other situations are commonplace and a great frustration for the would-be change agents in your company. Here are some of the common "failure modes" for transformation initiatives we encounter:

- The transformation never starts. As we discussed, this is the most common failure mode for digital transformation. It's easy for those in authority to see the threat as esoteric and hypothetical until the company comes under an existential threat—such as a digital giant moving into their space, or a rapidly-growing upstart eroding marketing share. Once the threat becomes real, the tendency is still not to transform, but instead to get into panic mode and address the most urgent symptoms ("We need a mobile [or cloud or AI or Blockchain…] strategy—NOW!") rather than the underlying cause.

  Investing money today to alleviate a threat or to capture an opportunity that might exist tomorrow is not easy for many businesses to do. However, it's the world around you that has transformed in the last generation; digital transformation of your business is simply the response required to recognize the new status quo. If you are still in the same lines of business with the same revenue streams as you had a generation ago—or even a decade ago—your business is at risk in the next three-to-five years. You need to consider how a true digital native could potentially disrupt or even eliminate your whole sector. Unless you take those disruptive actions first, you run the risk that someone may very well do it for you.

- Sabotage. Even when the company as a whole will benefit from a transformation, some of the individuals within the company perceive that they will not. They may be closely identified with a legacy product — or the legacy product and its technologies may be their primary skillset —and they fear losing their job or their influence if the next-generation product is successful.

  In some situations, people are happy with the current system because it is, frankly, an easy scapegoat for any sales or other numbers that fail to materialize. They are happy in their current role and fear the accountability that would come with removing the legacy system as an excuse. They may also fear being held accountable for the success or failure of a new system or new initiatives if they are seen as supporters. They may perceive it as safer to support incremental evolution of status quo. These individuals may attempt to sabotage the transformation project by publicizing flaws, by "dragging their feet" around deliverables, by continually raising objections, etc.

- Power grab. People who own a legacy product often have more influence within an organization than those initiating change. This is because the legacy is often responsible for the current revenue stream, and power tends to follow revenue. Seeking to own the new system may be productive if the person seeking ownership is a genuine supporter and is willing to "kill off" and replace the legacy they now own. However, it can also become a way to directly or indirectly suppress the future and maintain the status quo.

- Lack of confidence. Sticking with the "devil you know" is often seen as the low-risk option, even when there is a compelling need or when a new opportunity is there to seize. Some organizations are inherently conservative and may have evidence of past failures to support that conservatism. These people may be impossible to persuade due to their inherent conservatism.

- Shifting directions / project orientation. Many transformation initiatives are derailed by frequent changes in strategy. Even the fastest transformation approaches will generally take several quarters to execute. If the management strategy, commitment, or the management team itself changes more frequently than that, many things will be started but nothing will be completed.

  In addition, some organizations emerging from a strict IT-focused mindset are used to budgeting project-by-project rather than year-by-year, and this can lead to choppy forward progress as teams start and stop work. Choosing the right strategy for the transformation is essential to reducing the impact of changing direction.

- Transformation Approach. If never starting is the primary reason transformation initiatives fail, the most common reason those that do start end up failing is because you have chosen the wrong transformation approach for your situation. As we will discuss in the sections to follow, choosing the transformation approach that best aligns with your true internal and external situation is the primary factor that you, as a transformation agent, can control. Much more on this in the sections below.

These are just some of the factors that cause transformation initiatives to stall or fail. Transformation is a situation where real success requires that many things go right. However for many business, transformation is no longer an option. For these businesses, overcoming the obstacles to success is vital.

Summary and Conclusions

Summary and Conclusions

The focus of this paper has been on the economics driving your choice of a technical path to digital transformation. As we have seen, however, a successful transformation is not just—or even primarily — a technical activity. In particular, a transforming company needs to aggressively shed its past and fully embrace becoming the digital native version of itself — before someone else does it first.

To do this, a company needs to achieve clarity around its mission (i.e., the precise nature of the value it delivers to its customers, cleanly separated from the means currently used to achieve that end goal). The successful company then needs to determine how its unique value can best be delivered in a digital world. Then, and only then, is it truly positioned to transform.

This raises the question: Can any company be this smart? The honest answer is no. Neither Steve Jobs, nor Elon Musk, nor any business icon you care to name has total, unfaltering clarity about the details of their mission combined with a certain view of the future and of how to get there. The future is progressively revealed; even our heroes can only see so far ahead. At best, they see the next few steps as they, like us, fumble toward the future.

What the successful transformational figures do have, however, is a North Star.

Knowing where you want to end up — even vaguely and imperfectly — is the most powerful tool a leader can have in terms of taking the next step. It's when we look on each new technology or market shift as a chance to move a few steps closer to our ultimate goal that we begin to move in the right direction. Our progress will still be lurching and will contain many missteps. However, keeping the means opportunistic and the ends fixed (or mostly fixed) is the key to a successful transformation. It's when we get that inverted (i.e., mandated means and an unknown end-state) that things go haywire.

That's a little philosophical. What are the concrete steps to making a transformation successful? While there is no cookie-cutter approach, and definitely no deterministic linear progression, there are some activities common to those who successfully transform. Here are the steps we recommend to launch your journey in the right direction.

- **Define (or discover) the mission of your business.** Not an enumeration of your current business activities (those are the means), but a description of the value (the ends) you deliver, or aspire to deliver, to the people who are or will be the source of your revenue.

  Whole careers and advanced degrees are built around defining the vision and mission of a company, so we will not presume to teach such a skill here. In fact, in purist terms, we are actually mixing the notion of a "vision" and a "mission" in this discussion, and that mixture would not be well-received by such professionals. Nonetheless, there are many mission and/or vision statements that may be well-suited to other contexts, but are still not useful in guiding transformations. Here are a few thoughts about testing whether a given mission or vision can be effective as a North Star guiding you through a disruptive technology transformation such as digital:

  First and most important, your mission and vision for the future don't need to be right in detail—they just need to be "right enough." The details of your mission will change anyway as you head toward your North Star because you will know more, the future will be closer, and actively working toward a goal makes the goal clearer. You are better off starting to move in a roughly correct direction today than you would be waiting until you've clarified the details and wording of your mission many months from now. Transformation is one of those cases when you don't want to "make perfect the enemy of the good."

  Consider whether a "true" mission/vision statement already exists—perhaps informally—within your organization. Sometimes these are more powerful than the official ones, and therefore more useful for guiding transformations. These "true" mission statements are often discovered rather than crafted.

  For example, "[We] bring happiness to millions" was a saying in an early biography describing Walt Disney's personal hope for his work. Steve Job's description of the original MacIntosh computer as "insanely great" continues to motivate people at Apple (and Apple customers) today. Neither one of these is the "official" mission statement of their respective organizations, but as a North Star for waves of transformation, both have been extremely powerful.

Good missions are inspiring and steer the behavior of the organization. They are also "testable" in the sense that they are clearly met or not met by a real or imagined offering. (Is this insanely great? Will it make millions of people happy?)

Good missions transcend specific technologies and focus on the end value delivered to customers ("happiness," "insanely great products"). Avoid the "buggy whip" syndrome, and instead talk about the value you produce, not the means by which you happen to produce that value today. Otherwise you'll find yourself inventing a better buggy whip when the horse is no longer relevant.

The mission should not contain any words corresponding to the industry SIC code (or equivalent) for the business activities you engage in today. For example, if you are a bank there should be no mention of the word "bank" in your mission. If you are a file sharing service, there should be no mentioned of the words "file sharing." Rather, it should describe the value you seek to deliver (e.g. "protect wealth," "collaborative working").

The mission should be general enough to suggest additional lines of business beyond those you have today—whether you ever choose to pursue them or not. If it's not, you haven't thought deeply enough about the value you actually deliver—and it won't help in your transformation. "We bring happiness to millions," for example, is very suggestive and can be applied to any new technology. For example, "How can we use augmented reality to bring happiness to millions?"

- Consider how a startup would use the transformational technology (e.g. digital) to fulfill your mission. Don't think about how to make money from it yet.

  Again, the mission used in this step should not mention or enumerate the specific business activities you engage in today. Instead it should focus on the value you deliver to your paying customers. (Not "buggy whip" but "make transportation go faster.")

  This can be hard because to do it well, you need to forget about how you make money today. You also need think like you're looking in from the

outside of your current organization. In many companies this activity is not seen as a serious or valuable exercise by the "old guard pragmatists" who are focused on making this quarter's or this year's numbers with current products (or extensions to them), or on maintaining the relevance of their organization.

As an example let's consider the example of Kodak and Facebook from the previous section:

> — If we were a startup with Kodak's advertised mission of "sharing memories and moments," we would ask ourselves the question, "Using digital technology, what are all the ways we could share memories and moments?" That list would certainly include photo sharing—but it would also include posting text, enabling interactive chats and threaded discussions, a way to share items of interest or my own postings with various groups of friends having varying degrees of privacy, a calendar of events and activities you could share, and so on. It would also include "creating a moment" by playing games together, flagging the information that we like or hate and making it available to our friends for comment and discussion—and many other features.
>
> — On the other hand, let's say you did not think like a startup, but rather took the point of view of an established player—like Kodak. In that case you might ask yourself, "How can we use digital to drive more revenue into our photography business?" In this case, you'd come up with online printing services, in-store printing kiosks, etc.—exactly what Kodak did do. And, as we discussed in the previous section, we believe that mode of thinking is why Kodak is not Facebook.

If it's helpful, get outside help and facilitation in this brainstorming / imagining process. This is the single most crucial step in the transformation process: imagining where you want to go. The activity of visualizing future products and customer journeys for new services is often called "strategic design." There are a number of companies (including GlobalLogic) who offer strategic design services. Working with an outside firm can be helpful in this instance because of the range of

companies they have worked with, their familiarity with current trends, and the "outside in" perspective they can offer. These are perspectives that are often hard to get internally.

- **Still thinking like a startup, consider how to make money from the activities you brainstormed in the previous step, from the standpoint of a stand-alone pure digital player.**

  Try to avoid thinking of digital as a means of driving revenue to your current lines of business; when that type of revenue enhancement happens, consider it upside. Instead, think like a startup and assume that you need to make your digital system profitable as a stand-alone venture. You may or may not ever actually choose to make digital standalone—but to serve as a true transformation vehicle, stand-alone profitability of your digital offering should be possible.

  If it's not, you need to go back to the previous step and look at the value that digital is creating for end users. If they are not willing to pay (directly or indirectly) for the benefits provided by the new technology, then you may not be using it in such a way that it creates enough value. Again, think like a startup and don't be afraid to go back to the drawing board until your stand-alone digital value proposition becomes clear.

  Your "digital" revenue stream may be totally different in kind and in nature from your current stream(s) of revenue. For example, considered as a pure digital player that the bulk of your new revenue may come from advertising, subscriptions, or transaction-based fees rather than from sales of physical goods. Implementing such new revenue streams affects accounting, reporting, sales, marketing, and indeed every area of the company. While managing new sources of revenue is generally a good problem to have, the rest of the organization will need to learn to deal with them, too.

  Consider whether and how to potentially offset the cost of initial investment in your new system by generating incremental digital native revenue during development. You may or may not need to do this, based on your company's

appetite for investment and need for immediate results, but you should still consider how you could do it if you needed to. Your circumstances and the need or desire to generate near-term revenue will strongly influence your choice of transformation approach: Greenfield, Side-by-Side, Gradual Evolution, or some combination of the three.

— As we discussed above, incremental revenue should ideally be the direct result of the new business models enabled by your transformation, rather than solely from incremental revenue driven to existing lines of business. If your transformation initiative is not enabling new revenue streams, it's not really a transformation—it's an incremental upgrade. Conversely, if it does enable new revenue streams, you now have the basis for a true digital transformation.

— Real-life, stand-alone, venture-funded startups generally have two or three years at most before they need to either be profitable or to at least to demonstrate a clear path to profitability. Otherwise they are redirected or shut down, with their assets sold or recycled into another venture. As an example, Facebook is said to have shown a small profit in Year 2 but probably did not achieve sustained profitability for 5 years. Facebook did, however, show a very high growth rate from the outset in terms of the number of active users. They also had revenue nearly from Day 1, demonstrating the validity of their advertising-based revenue model. These factors combined, Facebook's path to profitability was clear to both early and subsequent investors.

- **While still staying in startup mode, architect the system you wish you had—the one that delivers on the "digital native" value proposition you have defined—as well as supporting your "digital native" revenue and business models.** You may not implement it all at once, but you should think it through.

  Follow Stephen Covey's maxim to "begin with the end in mind." Take off the constraints and approach the technical solution to your business needs like a startup would—from a blank whiteboard, choosing from the best available technologies. For now, forget about your legacy: legacy systems, legacy business models, current staff skillsets, etc. While architecting, think like a small sharp team starting from scratch with nothing but a great idea.

Your architecture should deliver your next-generation system in a maintainable, supportable, and highly adaptable fashion—meeting both your new business's functional as well as non-functional requirements for scale, performance, reliability, operational efficiency in deployment, security, CICD, etc.

Upgrading to a modern architecture and modern technologies should not be a goal *in itself*; rather your architecture and "technology stack" should be chosen to deliver business value and satisfy both functional and non-functional requirements with the lowest effort, shortest time-to-market, and least cost. Very often, the functional, non-functional, and business requirements mean that you end up using new technologies and a totally new approach. But that's simply because modern approaches and technologies are better than the legacy ones, even accounting for a learning curve. But modernizing solely for the sake of being modern is not the right motivation. Choose your desired future architecture and your technologies to serve the holistic set of current and future business needs, not the other way around.

At the same time, don't consciously or unconsciously design an architecture or choose technologies for political, attachment, emotional, or other "internal" reasons. Companies often get in trouble when they pick technologies and architectural approaches because they have relationships with a particular supplier, or because that's what they have used in the past, or because they have an (emotional) affinity toward or against a certain technology or approach.

While you should not use the new simply because it's new, by the same token, you should not use the old or (now) second-rate just because it's comfortable. You should fearlessly pick the best architecture and set of technologies that offer you the quickest and best path to meeting your business needs and functional / non-functional requirements, both now and in the mid- to long-term. That's what a startup would do.

- Come out of startup mode and select the transformation approach that meets your business constraints and best delivers on the objectives you've discovered above.

  Now is the time to factor in the reality of being part of a (possibly very large) company and having a legacy system and internal stakeholders to deal with. If you've done a good job on the previous steps, you know—technically and business-wise—where it is that you ideally want to end up. The task of the current step is deciding the best way to get there by factoring in the technical, staffing, and business constraints.

  The economic pros and cons of the various technical transformation approaches are covered in detail in this paper, but to recap:

  — The Greenfield approach is the fastest and overall cheapest way to make radical shifts in business models and technologies, but it does not deliver immediate business value and can have limited transformational impact on your current staff.

  —The Side-by-Side approach can produce the fastest time-to-market for a subset of next-generation features and is often the best way of testing new business models quickly in the market. However, it adds complexity and overall it ends up taking longer and costing more than a Greenfield approach to get to your desired end goal.

  —The Gradual Evolution approach is a measured, technically conservative approach to modernizing your current system. Of the three major transformation approaches, it takes the longest time to deliver the "stand-alone" / "startup" value. However, it potentially has the largest transformative effect on your current technical staff.

- Execute your selected transformation approach. If it isn't working, admit it and choose another.

  Companies that are successful transforming usually don't succeed on their first attempt. Apple didn't, and the company in our Greenfield case study didn't. In fact, few companies do succeed the first time they try to transform. As we've seen throughout this paper, the need to transform digitally—when

it occurs—is not just a "nice to have," but a literal existential threat to a line of business or the entire enterprise. In this context, it takes a lot of courage to admit the failure of one approach, pick up the pieces, and try a new one. But successful companies do just that. You must also act with that type of courage to follow their example. The best thing you can do when a transformation attempt isn't working is to recognize it, learn from it, and start over.

An image that comes to mind is that of the pilot of a wounded aircraft. The pilot heroes in the movies (and sometimes in real life) are the ones who keep trying: First they try Option A, then Option B, then Option C—until they either find an approach that works and the plane lands safely, or they don't. But they keep trying until they save their passengers, or until trying again is no longer possible. Only by acknowledging what doesn't work and trying one more time can they succeed. While digital transformation isn't a matter of literal life and death (thankfully), it can certainly be one for a company, and for the jobs and careers of those involved. This puts a lot of pressure on those piloting the change to claim and even believe they are being successful—even when they are actually in a tailspin.

How do you know when your approach is failing? It is certainly possible to fail to deliver a system that otherwise would have succeeded in the market; you can track that type of failure through normal software delivery metrics. However, the more usual failure mode is the other way around: Successful delivery of a system that meets internally-defined criteria, but not those of the real-world. We saw an example of the successful technical execution of a Gradual Transformation strategy, executed while the value of the company declined by over 90%. It's clear from this and similar examples that the success of your transformation initiative should not be judged primarily by how well it meets internal goals. While meeting internal project success criteria is clearly important, perfectly executing a plan is not good enough to say that your transformation is working. What counts more is whether your transformation is capturing the external market opportunity that has been created in your space by digital.

To assess the success of your transformation initiative, think like a startup and use the criteria that a savvy early-stage investor would apply. In some

cases, success can be measured directly. For example, in the Side-by-Side approach, you can measure the uptake and revenue of your new digital offerings. For other transformation approaches, your external deployment is "all or nothing" (i.e., Greenfield or Gradual Evolution). In these cases, it's often best to ship with a "minimum viable product" that is a subset of the full functionality and then elaborate. Until it's deployed, your external success during development can only be measured by sampling the enthusiasm generated in your stakeholders. These stakeholders include your investors, shareholders and their proxy, and your board. They also include customers and prospects, in cases where these initiatives can be made selectively or entirely public.

But "enthusiasm" is a proxy metric whose value is only as good as the information flowing both ways and your ability to deliver against expectations in a timely manner. What counts in the final analysis is getting your offering—or a subset of it—before actual revenue-generating users and beginning to reap the economic rewards of the transformation. If you aren't doing that, you need to constantly ask yourself if you're on the right track, and if there's a way to get there faster. Keep your "eyes on the prize"—which in this case is the revenue streams generated through the new business models enabled by digital.

Be sure your approach actually is failing before you try a new one. You may just be in a rough patch, and persisting will lead to success. Changing directions more often than you need to will just send you further away from your North Star. Keep in mind that you don't need to execute flawlessly to be successful. You just need to focus on delivering a new source of value to your customers by moving closer and closer to your "North Star." Test against it at every point.

Don't equate missing detailed internal milestones with failure. While your North Star itself needs to stay fixed, you don't need to keep your detailed internal objectives static to achieve a transformative end result. While it may shock you to hear this, "good enough" is often good enough, when what you are doing is delivering a truly new source of value to your end users. Think of any actual innovation in recent history: the iPod, the iPhone, the Cloud, Android, etc. In each case, the initial version indeed gave users a taste of the ultimate value they would receive. But in each case, the first version was not nearly as good as later incarnations of that innovation (or its successor) would be.

Sometimes changing some of the details around internal objectives will get you closer to your North Star, while keeping them static can lead to failure. This is a major reason why successful innovation companies often keep the details of their product roadmap private: So they can change them. This is not to excuse sloppy work, but in the immortal words attributed to Steve Jobs: "Real artists ship!" Waiting for perfection delivers no value at all to your end users. Through digital transformation, you are creating entirely new sources of value for your end users. This is not a minor enhancement of an existing system—this is something entirely new. In such a case, you almost invariably serve your users better by delivering part of this value sooner, even if that means compromising to some degree on perfection or functional richness.

— Insisting on keeping the details of internal objectives rigidly fixed is one way detractors of transformation initiatives try to sabotage them. In this tactic, opponents use purely internal criteria to structure a political failure, when in fact the transformation is advancing in the right direction to deliver external, holistic value to the company and its customers. Before opposing these detractors, though, be honest about whether changing the details of the end-state truly does serve the needs of the organization. The key question is "will this change bring me closer to delivering against my North Star in practice?"

— It's not because we're fans of sloppy work that we put so much emphasis on being OK with imperfection. Quite the contrary. It's because we have seen more companies paralyzed into inaction by a fear of failure, or fear of the unknown, than we have seen companies charging recklessly ahead. When the need for transformation heads toward you like an approaching train, you do not want to be a deer in the headlights, standing on the tracks. You are better off moving in any direction—ideally an approximation of the right one—than you are standing still.

At the same time, you need to apply this guideline responsibly. If you are making a life-critical medical device, the core system needs to work with full reliability before you can ship—this is not negotiable. Similarly, a core banking system must have complete reliability managing account positions and other critical information. What is negotiable, though, is the "bells and whistles"—the "frills" or "surrounding systems"—that are part of any software product, including life-critical ones.

Be rigorously honest about separating the literal MUST items from those that properly should be prioritized as SHOULDs and COULDs. Never compromise on the items that are genuinely MUSTs, but think hard about whether you deliver more value to the end user by delivering now without a given SHOULD or COULD. In a transformation scenario, where you are delivering genuinely new value, put your bias on timely delivery over delivering non-essentials.

A real disruptive transformation that changes or introduces whole new business models is difficult on many levels. Those challenges are technical, political, emotional, organizational, and more. If your initiative bogs down, or loses its champions, it may be time to change directions—or maybe it's just time to ease up temporarily. (See our blog on knowing when to back off for more discussion.)

- Shed the past.

Once you've successfully put your next-generation digital native system in place, it's time to commit to it and get rid of the old system it's meant to replace. Many companies stumble or falter here, and in almost every case the past holds on longer than anyone would reasonably believe it should.

The word "transformation" means, literally, to "change form." Taking on a new form requires shedding the form you have today. This can be exceedingly difficult, both for companies and for individuals. In both cases, the past made us what we are today, and its imperatives tend to drive our current behavior unless we replace those imperatives with others. For an organization, the past tends to become institutionalized and embodied in the political power structure of a company. For an individual, the same is accomplished by habits, learned behaviors, and other psychological mechanisms.

In both cases, shedding the past requires a conscious effort to switch our motivation from being driven by what's come before to being motivated to reach a future we can only imagine. This is scary, and fear (admitted or couched in the form of "business risk") is often the major factor holding us back. At an individual level, there's a saying along the lines of "Imagine someone stronger, smarter, and better than you. Then do what that person would do." While a little self-helpish, this saying provides a good example of consciously being motivated by the future —by your aspirations — rather than by your history and your current "form." For a business, let's take a look at our Kodak example as a case in point:

— Think back to the "digital native Kodak" hypothetical we described earlier. In that section, we posited that an aggressively transformational Kodak would have "changed form" into a Facebook-like entity. Indeed, had it been successful in transforming, Kodak itself would have become what we now know as Facebook. This transformation would have allowed Kodak to follow its North Star of "sharing moments and memories" as a true digital native. But let's think about what achieving such a change in form would have meant for Kodak in the late 2000s and early 2010s

— Specifically, this transformation would have meant taking the conscious decision to either divest Kodak's traditional chemical film and photography business, or to allow it to fail (as indeed it did fail historically). Kodak would have needed to shed its legacy entirely to allow the revenue streams flowing from the new Facebook-like digital native entity to support the growth of the new business. This is clear from the revenues that Facebook did require and the growth it did achieve.

— In any established business, power tends to align with those who have produced revenue in the past, as opposed to those who might, could, or will produce it in the future. While sensible from one perspective, it also means the Mark Zukerberg-like change agent at our hypothetical "digital native Kodak" would have been undermined and hampered from the start, no matter how talented he or she was.

Also, such a combined legacy-plus-digital-native entity probably would not have experienced bankruptcy because of the positive revenue growth generated by the new (fictitious) digital native activities. While the survival of the combined business is great in one sense, the very success

of the combined legacy-and-digital entity obscures the issue of the failing legacy business, leaving the viability of the traditional business open to discussion.

— Using the new revenue stream to prop up the old creates ambiguity around whether the legacy business is really failing or just in a temporary slump that can be rectified by some new strategy. This, combined with the inherent conservatism coming from the history and power balance being on the legacy side of the business, tend to steer businesses away from shedding the past. Instead, the internal criteria tend to motivate companies balancing legacy-versus-the-future to use the revenues from the future (digital) to try to enhance or sustain their traditional revenue streams. In other words, they "feed the past and starve the future."

In our hypothetical digital native Kodak, as we can see from the real Facebook's success as a stand-alone entity, this would have been exactly the wrong thing to do for the combined hypothetical Kodak's future prosperity or even survival. New revenue streams need fuel to maximize their growth. As their digital lines of business achieved success, the right thing for our hypothetical company to do would have been to let the past die with as much dignity as possible, while shifting their focus to digital.

- Begin shifting your business models to "digital native."

  Once you've shed the past, you need to cut the lifeline and fully commit to the future. Instead of harvesting your new revenue streams to sustain your legacy business, you need to do the exact opposite: cannibalize your current revenue streams and/or your continuing investment in your legacy and instead fund the future. In other words, shut down or divest the past, and fully commit to the future. This is extremely difficult because it means totally upending the current power structure of your business. However, letting the future become hostage to your past is not a winning strategy. You need to let it go, and commit to the future you have chosen.

A transformative technology disruption on the order of "digital" is a perfect storm of events that happens a few times in an entire career, and not often in the lifetime of a company. Probably the last time anything comparable to "digital" happened (where under that umbrella we include related technologies) was the

introduction of the "personal computer" in the 1970s and 80s. By comparison to "digital" as it exists today, even the public availability of the Web, email, and other new forms of communication in the 1990s was just a prelude. Looking back now, the Web was only the beginning of the perfect storm that digital has become.

The way we live and work has changed profoundly in just over 10 years. It's hard to even imagine it, but the now ubiquitous "smart" mobile devices that are now so central to our lives, such as the iPhone and Android, did not even exist until the late 2000s—little more than a decade ago. Coming with and supporting that revolution was a parallel one in technologies, such as the Cloud, NoSQL, near-zero storage cost, low-cost and ubiquitous connectivity, open source, containers, event-driven architectures, practical AI/ML/NLP, computer vision, reasonably good speech recognition, and many others.

The net effect has been to transform the basis of how people interact with their tools, and their environment, and each other. These are the fundamental underpinnings of business, commerce, and nearly every other human activity. It is no wonder that digital is causing such profound disruption—and creating such enormous opportunities.

For an event that is so rare, it is completely normal that we don't initially understand how to deal with it. Like being a first-time parent, it's a miracle if you get things right the first time—and in fact it almost never happens. In every case, the best attitude we can take is to expect to make mistakes, learn from them, fix them, adapt, and grow. No matter how hard you try to do it perfectly, you won't. Learning from and addressing our mistakes is how we ultimately succeed in any once/rare-in-a-lifetime endeavor.

The actionable activity is continuing to separate the "means" from the "ends"—that is, to clarify the mission of your company in terms of the value it delivers to your customers, rather than its current activities. Then fearlessly determine if a "technology native," unconstrained by the past, could use new technologies to deliver that value more effectively. If the answer is "Yes," then act with the vigor of a startup (or else buy and nurture an actual startup). Monitor success and either change course if it doesn't work or—if it does work—aggressively shed the past before it drags you backward.

In particular, one economic lesson we've seen play out many times is that letting internal factors alone determine how you deploy your assets will, except by luck, almost certainly lead to failure. In bet-the-company situations, it can lead to the literal failure of the entire enterprise. We've seen a real-life example of this in the Gradual Evolution case study. A take-away is that an approach that seems risky in the context solely of internal considerations (company history, culture, existing skill sets, etc.) may actually be the least risky option when you look at the company situation holistically and strategically—both internally and externally.

When we make economic decisions—whether they are personal or business—our emotions and attitudes surrounding risk can be a major obstacle to making good decisions. In particular, letting go of the past is the biggest barrier many of us face. To make a good decision, we need to ruthlessly face the situation that exists today, and take our best shot at projecting the opportunities and challenges the future will bring. That's the best any of us can do at any given time. While we can learn from the past, reluctance to let go of it—whether it is a fear of taking a loss in a bad stock, or being willing to shed a code base that no longer suits our needs—is often the hardest part.

The more your strategy decisions are based on emotions rather than on the true, holistic economics of your situation, the more likely it is that you'll start on the wrong path. Attachment to an existing code base because of the effort it took to create it, the hidden belief that to throw it away would mark it as a "failure," and maybe some inertia and fear of an unknown future with a new set of challenges—these psychological factors are powerful, and ones that we can probably all relate to in some areas of our lives. But they are the wrong basis for deciding because they all pertain to the past or to our fears, rather than the present as it truly exists.

At a company meeting, Steve Jobs once said that looking back over his career, his biggest failure was holding onto the past too long. And this is Steve Jobs—almost universally seen as one of the most fearless innovators in the last (or early 21st) century. That the rest of us share the same tendency is nothing to be ashamed of—but it is something to be acknowledged and overcome, just as he did.

Digital transformation is a challenge only faced by successful companies. Companies that have NOT been successful do not have the option of transforming—instead they get acquired, or go under before they have the opportunity. A history of past success is a great legacy but can also become a challenge. Our normal, human tendency is to believe that since you've been successful in the past doing X, you will continue to be successful in the future doing the same thing. This assumption is no longer true when external conditions have profoundly changed. The more internally focused the decision-makers are, and the more removed they are from the reality of external conditions and technical possibilities, the more likely their choice of transformation approach will be the wrong one.

Another challenge decision-makers face is having a limited view of success. As structured and intensely goal-directed people, we engineering executives are especially prone to this type of myopia. Again, looking back to the Gradual Evolution case study, while we were not involved in the implementation, our feedback from people who were involved is that the actual mechanical process of transformation itself was seen internally as a success!

That is, the code base was indeed refactored, and the technical goals were met—on time and (to the best of our knowledge) on budget. Certainly, bonuses would have been payable all around—had not the value of the company declined by over 90% while the strategy was being executed. One is reminded of the cynical adage: "The operation was successful, but the patient died." The point is that our view of success—the goal of transformation—must be holistic enough to encompass the commercial success of the whole company or line of business, not just that of a department, team or project.

Let's consider for a moment who was at fault in the Gradual Evolution case study—not to point the finger at anyone, but to prevent a similar situation from happening to you. Was it the engineering organization's "fault" that they chose what turned out to be the wrong transformation approach? Granted, engineering was indeed too conservative, and a more aggressive engineering champion might have changed the outcome.

However, one maxim in business is that the blame for the loss of a given amount of money lies with the people who are authorized to write a check for that

amount. Giventhat this company lost more than 90% of its value, I doubt if any SVP / EVP of Engineering on earth has that proportionate level of spending authority. The blame therefore lies at the top of the company—with the Board and the C-suite.

Just as a famous statesman once said, “War is too important to be left to the generals,” making transformation decisions is too important to be left to engineering alone. This is a strategic decision whose pros and cons will reverberate at every level of your organization.

Perhaps the biggest challenges to choosing the best transformation approach are the emotional ones: in particular, a fear of failure or even a fear of a success that brings decreased status or even irrelevancy to the players initiating the transformation. Fear of failure is something most of us can relate to, and it can be particularly acute when we go down a path we have never tried before. That is the case for most transformation initiatives, since the compelling need for transformation is a rare event in any career. The best advice is to not make decisions out of fear; instead choose what is best for your company holistically.

The fear of success may seem a bit strange and can be more insidious. As an example, with one client we proposed a next-generation architecture that dramatically simplified a major element of their existing system. This simplification was possible both because of new technologies and because the new architecture paradigm enabled by those technologies let us eliminate most of the complexity of this subsystem while still delivering improved performance and functionality. The current owner of the subsystem, who was in the room for the architecture discussion, looked shocked and said, “But if we do that, the size of my subsystem would be less than a third of what it is today. What do I do with the rest of my team?”

Far from being a dumb question, this statement was refreshingly honest—and we appreciated the owner getting the issue out in the open where we could discuss it. What more commonly happens is that those threatened by success either subtly subvert the transformation process or, if they have the authority, give lip service to transformation but never productively begin it in the first place.

Fear of success and its known or unknown impact on an individual decision-maker or influencer drives much of the politics and poor decision-making that surrounds many would-be transformation initiatives. It is more comfortable to rearrange the deck chairs on the Titanic than to acknowledge it is sinking, especially when the situation is totally outside your experience and you don't have the first idea how to prevent the coming disaster. We very much hope that this paper has given you some indication of how you can at least start to keep your ship afloat and make headway.

In conclusion, the economics of digital transformation compel you to take a clear-eyed view of both your internal and external environment: the "internal" so that you know what you can realistically accomplish given the resources available to you and the reality of your organizational dynamic; and the "external" so you understand the goals necessary to achieve business success—not just the success of your project. Given these insights, you can craft the best technical means to get there—whether that is Greenfield, Side-by-Side, Gradual Evolution, or some variant of all three.

About GlobalLogic

GlobalLogic is a leader in digital product engineering. We help our clients design and build innovative products, platforms, and digital experiences for the modern world. By integrating strategic design, complex engineering, and vertical industry expertise — we help our clients imagine what's possible and accelerate their transition into tomorrow's digital businesses.

Headquartered in Silicon Valley, GlobalLogic operates design studios and engineering centers around the world, extending our deep expertise to customers in the communications, automotive, healthcare, technology, media and entertainment, manufacturing, and semiconductor industries.

www.globallogic.com