White Paper

Modernizing Business Applications

by Dino Esposito

wisej.com

Wisej.NET

White Paper

Table of Contents

CHAPTER 1

The Software and Business Bond

> *Software eventually and necessarily gained the same respect as any other discipline.*

Margaret Hamilton

Author of the software for the lunar modules of the Apollo missions

As surprising as it may sound at first, software, as we know it today, was never intentionally planned in the history of humanity. Throughout the centuries, polymaths explored mathematics, logic, physics, and philosophy. Building on these studies, in the 1940s, the first electro-mechanical machines were designed and built, giving spark to what we ultimately came to call "software."

None of the prominent figures in computing, including Turing, von Neumann, Kilburn, Shannon, and McCarthy, were envisioning anything remotely resembling today's software. They had, instead, a much more ambitious and staggering goal: the mechanization of human reasoning. The early "thinking" machines of the 1950s were only massive hardware structures of valves and pistons brought to life by electricity. Software only gained its own significance and identity towards the end of the 1960s. In fact, the term "software engineering" was first reported in the mid-1960s. No one set out to create software; rather, it emerged as a byproduct—if not residual waste—of more ambitious research (that, by the way, failed at the time with unsuccessful outcomes).

Just the same two words that today spark discussions and make eyes gleam—Artificial Intelligence (AI)—seventy years ago were coined (by John McCarthy) only to unify under the same conceptual roof two distinct research paths trying to mechanize reasonings: cybernetics and automata. The sacred fire of AI remained inert under the ashes for decades, extinguished by the practical impossibility of implementing algorithms that followed the computational model of human neurons.

Since the moon landing era, beyond early futuristic machines, the project of AI has remained a rudimentary dream shattered during tests, but spreading around debris of apparent little appeal but destined to gain respect and power—the software.

In the Beginning It Was COBOL

The first successful execution of a computer program occurred in the summer of 1948 at the University of Manchester care of Tom Kilburn and Alan Turing. At the time, though, researchers were still seeing computer programs as the necessary means to reach mechanical reasoning. Over the successive years, however, as the ultimate goal drifted further away, it became apparent that the "soft" part of a "hard" machine could effectively guide valves and transistors in executing business and scientific operations.

Interestingly, no actual FORTRAN code was used in the control systems of Apollo rockets. Instead, the code was effectively developed and tested using a FORTRAN dialect called MAC. Once the design was finalized, the corresponding assembly language code was written on paper and then manually woven into the memory circuits.

The FORTRAN programming language created by John Backus at IBM in the 1950, played a crucial role in the Apollo missions. In particular, engineers used it to write extensive code for analyzing data from rocket engine test firings and understanding critical aspects like combustion performance, chemical reactions, heat transfer, and other vital rocket processes.
If the FORTRAN language was designed with the noble purpose of aiding in intensive scientific calculations and, to some extent, advancing mankind, another language extremely popular in the 1960s took on the less honorable role of primarily advancing business—the COBOL language.

COBOL (short for Common Business-Oriented Language) emerged in the late 1950s as a programming language designed for business applications. It aimed to be English-like and easy to read, facilitating the development of applications for commercial data processing. COBOL was specifically designed to run on mainframe computers. Mainframes are large, powerful computers optimized for handling massive volumes of data and transactions. COBOL programs were used extensively on mainframes to process and manage vast amounts of data efficiently and quickly became the backbone of many businesses and government organizations for their line-of-business operations such as accounting, payroll, and inventory management. A line-of-business (LOB) application is therefore a piece of software specifically designed to support and automate the activity of a particular department (or line of business) within an organization.

COBOL and mainframes have a deep-rooted relationship as if each was tailormade for the other. Despite the rise of newer technologies, many organizations still today rely on mainframes running COBOL applications. The longevity of COBOL and mainframes is largely due to their proven track record of reliability, scalability, and security.

Eternity Is Not an Attribute of Software

Despite being hard-coded in some servers and operating relentlessly 24x7, software is an incredibly dynamic and alive entity. As weird as it may sound, software is subject to the action of time like any other living organism.
The duration of software viability depends on various factors such as technology advancements, evolving business needs, and maintenance efforts. Basic applications may last a few years, while complex systems can remain relevant for decades with regular updates and checks. In general, critical line-of-business applications typically have a lifespan of 5 to 10 years before the question of significant modernization or replacement arises. Legacy software may persist longer but becomes increasingly costly and risky to maintain.

Twenty years of software life marks the delicate threshold where longevity and obsolescence come into sharp contrast. By thirty years, action can no longer be delayed, and a decision—any decision but the right one—must be made.

The Mechanics of Software Changes

Speaking in more general terms, the critical age of a line-of-business application varies depending on

several factors, including its complexity, technology stack, maintenance history, and business requirements. By rule of thumb, a line-of-business application typically becomes critical when it reaches a stage where the functionality is essential for daily business operations and any disruption or failure would significantly impact productivity and revenues of the organization.
As mentioned, this can occur anytime in the first decade after the initial development, but it ultimately depends on how well the application meets the evolving needs of the business and how effectively it can adapt to changes in technology and market demands.

The first COBOL applications deployed starting from the 1970s were no exception.
Executives soon realized that completely replacing a well-functioning (albeit outdated) COBOL and mainframe application was undeniably daunting and challenging. However, compelling reasons emerged to move beyond mainframes and the COBOL language. For executives, striking a financial balance between maintaining old-but-functional systems and investing in new-and-forward-looking technologies has been a recurring dilemma since the 1990s.

From an executive perspective, what are the triggers for actual change? It's mainly two of them.

Technological obsolescence. A system doesn't become outdated alone but the advent of newer and superior technologies makes it legacy to the point of hindering competitiveness.

Market disruption. New competitors entering the market, shifts in customer preferences, or just new phenomenal business opportunities, require new products and services.

There might be more to it, such as regulatory changes requiring companies to comply with new standards, changes in leadership and strategic direction, and financial considerations like cost reduction initiatives or the need to invest in innovation. In short, though, when a new superior technology emerges or a business breakthrough occurs, action is required.

For executives managing COBOL and mainframe applications, the time to act came in the mid-1990s. COBOL was the first widely used language for business purposes, but others appeared in the decades that followed, up to the crucial millennium transition.

The Promised Land of Java

Up until the mid-1990s, software development faced several challenges due to the limitations of existing programming languages and platforms. A few of them in particular:

Platform dependency. Most programming languages at the time were tightly coupled with specific hardware and operating systems. Hence, any software written for one platform often couldn't run on another without significant modifications.

Lack of interoperability. Integrating software components from different vendors or, worse yet, written in different programming languages was highly problematic due to neatly separated runtimes, compatibility issues, and lack of standardized communication protocols.

Limited expressivity. Developing large-scale software systems was complex and error-prone as programming languages often lacked features for modularization and abstraction resulting into spaghetti-code and maintenance headaches.

In a nutshell, the software development landscape of the early 1990s appeared fragmented, as developers struggled with platform dependencies, interoperability challenges, and even uprising security issues. And then Java came along.

Facts of Java

Java was created by Sun Microsystems around 1995 and subsequently acquired by Oracle in 2008. It is a versatile and platform-independent programming language known for its simplicity, reliability, and security. It fully supports object-oriented programming, provides automatic memory management, and showcases an extensive range of libraries and frameworks.

Java gained wide acceptance in the software industry because it specifically addressed many of the software challenges of the time and, to a large extent, revolutionized software development practices. In a decade, Java took a crucial role in the transformation of software-led business operations.

Java's key strength lies in its platform independence. Java programs, in fact, can run on any device that has a Java Virtual Machine (JVM), making it an ideal choice for building applications that need to operate on diverse hardware and operating systems. This portability made it possible for businesses to develop software once and deploy it across multiple platforms with limited extra effort.

In mission-critical scenarios, Java's robustness ensured the smooth operation of essential applications and the extensive ecosystem of libraries and frameworks further enhanced its capability to address complex business requirements. Last, but not certainly least, Java's object-oriented nature promoted modular and maintainable codebases, collaboration among development teams and seamless integration between new and existing systems. Furthermore, Java has been the first language to introduce a sandbox environment and built-in cryptography making it a trusted choice for applications handling sensitive data.

Java and Business Applications

It's roughly estimated that during the mid-1990s, a significant portion of new application development, possibly around 20-30%, shifted to Java from COBOL or other languages. Also, a considerable number of existing (and old-enough) platforms were moved, in total or in part, to Java. To cut a long story short, at the turn of the new millennium, Java definitely had become a cornerstone in reshaping line-of-business operations. Today, Java drives the backend of numerous large corporations, particularly in the banking industry, and is second only to JavaScript in terms of developer popularity. As of 2023, it's estimated that around 17 million developers worldwide use Java forming the oldest and most active developers' community in the software space.

One of the keys to Java's success is its inherent budget-friendly nature. In about three decades of development, Java has continuously improved its performance across different environments, be it desktop or mobile. One notable feature is its backward compatibility, allowing code written in previous

versions to seamlessly work in newer ones, minimizing user friction and the need for extensive code refactoring. This stability has made it a preferred choice for long-term commitments, especially for traditional and change-resistant organizations like banks and government agencies. For the same reason, also newer but established corporates opted for Java in their backends; among the others Google, Netflix, Spotify, Uber.

In this regard, Kotlin offers substantial support for keeping Java investments compelling and modern. Introduced by JetBrains in 2016 and quickly adopted by Google for Android Studio development, Kotlin boasts its own syntax full of modern features such as null safety, smart casts, data classes, extension functions and coroutines. It was also designed for conciseness, tackling Java's structural verbosity that often leads to excessive boilerplate code. What's particularly interesting about Kotlin is that despite its different syntax, it compiles to the same JVM bytecode as Java applications, ensuring full interoperability between modules written in these two languages.

Since the early days, the perfect fit for Java as a programming language and an all-round development platform has been the corporate environment. It was for the reliability and maturity of the solution, the extensive ecosystem of companion frameworks and the slow, though steady, evolution. In a way, Java represents for established businesses a choice made once and for all.

In the 1990s, many corporations transitioned to Java and wholeheartedly embraced the new platform, making a long-lasting choice that still stands today, three decades later. Furthermore, the recent emergence of Kotlin ensures that Java investments remain strong, and a future where Java code gradually transitions to Kotlin over the next few decades is, in many ways, feasible. As for the tooling, JetBrains, a tool and language company, supports and fosters the Java community with top-tier IDEs and development products.

However, what about the companies that didn't initially adopt Java?

25 Years of the .NET Revolution

In 1999, hardly anyone outside a select few offices within Microsoft's headquarters in Redmond knew what was happening. One day a small group of people from the fast-growing technical publishing industry was invited to an insider's meeting in Redmond. Before heading to the meeting, one of them asked me about "my" personal hopes and expectations for the future of computing around the Microsoft's Windows stack. It was a shocking kind of questions, but I well remember I took it seriously!

If I were the god of Software, what would I do to empower developers?
Then I envisioned a world of system-provided objects encompassing the entire functionality of the underlying Windows platform. Additionally, these objects could be programmed in the same manner across various programming languages. Furthermore, I imagined a new programming language easier to use than C++, less verbose than Java, and hotter than Visual Basic. Lastly, as icing on the cake, I dreamt of a built-in framework rich of ready-made objects (lists, collections, database, and network tools) for everyday coding.

When my friend returned, constrained by a strict NDA, all he could say was, "Dino, you're going to go through a lot of good things in the very near future."

The "good things" my friend was referring to was the upcoming Microsoft .NET platform and ecosystem.

The .NET Big Thing

Microsoft .NET made its official debut in 2002 as a framework for building Windows and web applications. It shared many similarities with the Java ecosystem, including side libraries, an underlying virtual machine, and a meta language in between programming instructions and assembly code. However, two crucial differences set it apart: the introduction of a new and well-thought-out programming language C#, and support for a wide range of alternative programming languages. A decade after its initial release, .NET underwent a significant overhaul—.NET Core—evolving into a cross-platform ecosystem that now supports Linux and macOS alongside Windows. Later on it also integrated deeply with the Azure cloud platform and recently with the OpenAI platform for artificial intelligence solutions.

.NET and Java collectively dominate nearly 100% of worldwide development activity. Despite recent improvements that have expanded its market, .NET has been a significant milestone in computing since its inception, greatly impacting the field of Information Technology. The key factor behind .NET's rapid adoption was its provision of a unified platform for building various types of applications, including desktop, web, and mobile.

Technically, .NET moved away from the complexities of C/C++ based Windows programming of the past. It introduced features like managed code execution, garbage collection, and a rich class library, making development and database access easier, more efficient, and less error-prone. Its web platform, ASP.NET, revolutionized web development by enabling the rapid (and partly visual) creation of applications, initially without even requiring extensive knowledge of web tools and HTML dialects.

Overall, .NET simplified development, reduced time-to-market, and facilitated the creation of robust software solutions for various industries. Those who hadn't yet adopted Java found, a few years later, an even more compelling alternative.

.NET and the Web Breakthrough

The timing of .NET's release couldn't have been better. It coincided with a major industry breakthrough: the emergence of the web and, more importantly, e-commerce. Just as companies were struggling to adapt to the web, Microsoft introduced a compelling new

development platform that was highly productive and user-friendly. For those still relying on Visual Basic frontends connected to mainframes and large database servers, .NET provided a much simpler yet extremely powerful alternative to achieve "exactly" what they were planning to achieve.

The ASP.NET platform dominated the vast majority of new and revamped applications.
The original ASP.NET platform, known as Web Forms, was built and then marketed based on a perfect blend of facts and buzzwords. With a powerful server-side engine, ASP.NET Web Forms offered a comprehensive library of web controls, allowing developers to maintain the familiar client/server development experience. The successful slogan of the early 2000s was "You don't need to know HTML and JavaScript to write web applications," reflecting the true reality of the time. This era saw nearly every developer become a web developer, including those who had previously focused on other technologies such as C/C++, MFC, Delphi, Visual Basic and even COBOL and Java.

Today, many software component vendors acknowledge that a substantial portion of their revenue still comes from products directly, or closely, associated with the now partially deprecated ASP.NET Web Forms platform. There's a rationale behind this, and comprehending it is crucial for planning the optimal approach to modernize today's line of business applications.

In the early 2000s, ASP.NET presented a golden opportunity to rewrite and modernize business applications that were built with technologies that were at least 30 years old. It was a perfect convergence of factors: the need to modernize, new business opportunities offered by e-commerce, and a development platform that could accommodate developers with any background and of all levels of experience. What occurred immediately after the advent of .NET was not an isolated event but rather the reemergence of a long-term pattern: at some juncture, business and technology must undergo radical change. ASP.NET Web Forms embodied this transformation, which keeps it highly relevant still today, as we stand on the cusp of another pivotal industry breakthrough—the one driven by AI.

Microsoft itself seems to acknowledge very well the continued importance of older ASP.NET versions used in today's businesses. While they shortened support lifecycles for newer .NET Core releases (like .NET 8 with 3 years of support ending in 2026), older frameworks like .NET Framework 4.8 (released in 2019 and recommended for any up-to-date ASP.NET Web Forms application) still have indefinite support. This creates an interesting situation where recent .NET releases are unsupported sooner than much older technologies.

CHAPTER 2

State of the (Software) Union

> *Technology is always changing, but the key to success is not to focus on the technology itself, but rather on how you use it to solve real-world problems.*

Scott Guthrie

Executive VP, Microsoft Corporation

In the past chapter, we remembered the birth of software as a discipline and outlined the major achievements of the industry since the early days. There is a set of priorities that drive the progress. It's interesting that the fundamental drivers of the software industry have remained relatively consistent, although their relevance may have varied over time.

In a hypothetical speech over the state of the (software) union, we would identify the following top priorities and use them to address the current state of the industry, calls-to-action, and future directions.

- Problem solving and related level of customer experience
- Security of data and people and related quality assurance
- Innovation to stay ahead of competitors and meet evolving customer needs
- Scalability to accommodate growing user bases and data volumes
- Compliance to regulatory requirements and industry standards
- Interoperability for seamless integration with other systems and data exchange

More recently, a couple of other factors have become increasingly significant: the capacity to attract and retain talents, and emphasis on sustainability to minimize environmental impact and foster social responsibility.

Side by side with these broad priorities, there are numerous existing applications that, while solving critical problems, require continuous improvements in security, innovation, scalability, and interoperability. Nearly all executives today face the Herculean task of balancing these needs with budgetary constraints.

Legacy Applications

As tech professionals, every day we run across remarkable stories of innovative frameworks that rapidly conquer new development territories, only to leave them in a state of degradation just two or three years later.
Alongside these groundbreaking (and sometimes just ephemeral) frameworks, a significant portion of legacy applications thrive on sturdy frameworks that have stood the test of time.

Anatomy of a Legacy Application

Abstractly speaking, the term legacy software application refers to an older, often outdated, software system that has been in use for a significant period of time within an organization. These applications are typically critical for the business operations and have been developed using technologies, frameworks, and methodologies that may no longer be considered modern. Legacy applications may have been developed decades ago and may have undergone multiple updates and modifications over time.

The primary factors that classify a software application as legacy software include its technology stack and architecture, as well as challenges related to evolutionary maintenance, integration, and security. Legacy applications are often built on a monolithic and poorly modular structure, making them challenging to scale and maintain without risking uncontrolled

regression. Maintenance challenges mainly stem from outdated documentation, no longer supported software components, complex, difficult-to-read code bases and, overall, limited developer expertise in working with older languages and libraries.
Despite being legacy systems, most line-of-business applications continue to be vital for many organizations, supporting essential business functions. Modernizing or replacing these applications requires a significant investment of time and resources and is inherently risky, with a high potential for failure.

In the realm of enterprise legacy applications, .NET and Java dominate, and it is factually irrelevant which one is leading. .NET emerged at a very favorable juncture in the late 1990s when mainframe applications were ripe for redevelopment, and the rise of the internet made modernization appealing and feasible to all decision-makers. At the turn of the millennium, though, web applications were still largely viewed with skepticism so many organizations embraced .NET and Windows Forms to continue developing desktop applications in a modernized client/server style.

Most legacy .NET applications are Windows Forms or ASP.NET Web Forms applications whose core was built over a decade ago. There are also some indications that Visual Basic 6 (VB6)—a technology even older than the first .NET Framework—is still surprisingly alive. Microsoft, in fact, will continue to offer extended support for the VB6 runtime environment for the whole lifetime of Windows 10 and 11, indicating a continued user base. This means that VB6 will be around for about another decade!

Taxonomy of Legacy Applications

In the current business landscape, advancements in core technology pale in comparison to the potential changes that generative artificial intelligence may bring within the next two or three years, well surpassing the current copilot-style application extensions. Thus, it's that time of history again where the idea of upgrading legacy applications appears plausible even to budget holders.
Legacy applications can be classified in three main categories, each based on a variety of differently aged technologies.

Class of Application	Major Technologies
Desktop	Java (using Swing, AWT) Windows Forms (using .NET Framework) Windows Presentation Foundation (WPF) Visual Basic 6
Web	ASP.NET Web Forms Java Server Pages PHP
Mainframe / Client-Server	COBOL PowerBuilder IBM z/OS

It's worth mentioning that several other popular technologies from the past have already been phased out and replaced. However, some of the replacement technologies are now facing the challenge of time themselves. The list of buried technologies that may still be found around includes MFC (Microsoft Foundation Class) applications, ASP Classic applications and frontends, and form-based applications running on top of database systems. Finally, the list also includes older (often very old) versions of CRM platforms like Salesforce or SAP.

At any passage of time when the need for change is strongly perceived, a good number of these legacy applications disappear only to reappear in a new guise later. Sometimes with great success; sometimes with only a lot of money wasted. At the very end of the day, any (modernized) application inevitably will become a legacy application in a matter of years. Albeit the software may seem like an immutable and lifeless entity, when immersed in the context of the real world, it is in every respect a living organism and, as such, subject to the action of time.
The more you run it, the more is software getting older. That's because the surrounding business and technology landscape changes.

What would be a valid method and timing for upgrading such line-of-business applications? There are a few common reasons for the need of change. Any strategy should address at least a few of them.

Reasons for Modernization

Application modernization takes place with the purpose of breathing new life into business technology and accelerating digital transformation. But modernization initiatives that deliver the most value encompass many elements. Here are a few, quite common, ones.

Cloud adoption. Cloud hosting and computing allows for scalable, and often cost-effective solutions with improved accessibility, easier cross-organization collaboration and seamless integration with other cloud services. Even without reaching the peaks of flexibility and ease of deployment of cloud-native applications, to grasp the crucial importance of cloud adoption, simply imagine the effort required to update a desktop application within an organization compared to updating a web application.

Modern user experience. Much more than two decades ago, user experience (UX) is a discipline with its own practical rules, visual expectations, patterns and experts. Any modern applications must prioritize user-friendly standards for the interface more than they did only a decade ago. Worse yet, such standards evolve quite rapidly and ideally any application is expected to provide a relatively modern UX and certainly not a vintage one! Pillars of a modern UX are a responsive design, lots of personalized dashboards, and more intuitive interaction methods (e.g., drag-and-drop, touch, notifications, background tasks). For example, in a pure web scenario, a full page refresh after completing a form, while it may still be functional, is today unacceptable from a user perspective.

Security and compliance. Cybersecurity threats were not a major concern of applications devised ten or more years ago. In addition, regulatory requirements (e.g.,

data protection and retention, age restrictions, legal jurisdiction) and compliance with newer and sometimes stricter standards (e.g., accessibility, payment processing, security of the digital environment) have become paramount just for any business applications.

Scalable architecture. Scalability is a term that appeared in the industry at the time the web exploded. Since then, any application of any size is marketed (internally within the company and/or to the outside world) as infinitely scalable and scalability is mandatory to have in the list of nonfunctional requirements for just any project. Overall, what companies really look for is not a ready-to-use scalable architecture, but an architecture that provides a solid foundation for sustainable growth, operational efficiency and is giving them a competitive edge in what is and remains a fast-evolving digital landscape. A scalable architecture can't be improvised but it is quite expensive too. Yet, companies wish to be ready for it.

DevOps practices. To be ready to scale, cloud adoption is crucial but so is DevOps, too. Since the mid-2010s, the push for faster software delivery and increased agility in response to market demands has fueled collaboration and efficiency between development and operations teams. New development practices such as continuous integration, deployment pipelines, containerization, and test automation gained adoption and are now indispensable assets on top of any code repository. In particular, DevOps pipelines facilitate shorter development cycles, allowing for more frequent updates and improvements. In the end, more automation decreases the need for manual intervention and leads to fewer errors (that are easier to correct) and lower labor costs.

Data analytics. Yet another effect of general cloud adoption is the availability but, in a sort of vicious/virtuous circle, also the demand of more and more data points. The cornerstone of any modern business strategy, data analytics involves examining datasets to draw conclusions about the information they contain. By leveraging analytics, businesses can make smarter decisions, foresee challenges, and capitalize on opportunities, ultimately leading to sustained growth and success. Like scalability, it's another pillar to sustainable growth and to maintain a competitive edge in their respective markets.

Interoperability and integration. As companies merge to grow bigger or just survive, interoperable IT systems are crucial for seamless operations across different IT infrastructures, applications, and data sources. The challenge here is setting up an operational plan to address compatibility issues, standardize protocols, and ensure data migration and integrity. Use of ad-hoc APIs, middleware, and an architectural effort aimed at identifying standalone modules and prospective facades is the key to successful integration. Having an application that lends to integrate, and be integrated, is a sure sign of success.

Add to all these factors, the most intriguing, yet challenging, of all: AI in general, and generative AI in particular.

It's unpredictable today if the day of AI singularity will ever come or how long it will take. The current stage of AI technologies—albeit purely statistical—is powerful enough to accelerate users in the execution of routine tasks while making the work to be done less boring. More importantly, generative AI raises the

level of communication between users and software, jumping from rigidly structured forms to the free text of human interaction even over the barrier of different languages. Finally, more canonical neural networks—these days also a sort of legacy (sic!) AI—for prediction, classification, and recommendation tasks are close to be a commodity.

So, there are many valid and concrete reasons for modernization. What could be a successful strategy to it, though?

Overall, developing a modernization strategy to overhaul a critical software application is akin to creating a personalized workout plan to regain decent physical fitness. For it, we can even coin a new term—**software fitness**.

In personal fitness, an effective workout plan is tailored to the individual needs and preferences and takes into account factors such as the current fitness level, any injuries or limitations, and the types of exercises you enjoy. In software fitness, a general modernization strategy is based on three pillars, as in the synoptic table below.

Software Fitness	Personal Fitness
Stop new development	Stop bad life and nutrition habits
Isolate parts to work on leaving the rest intact	Target specific body parts without causing harm to others
Test coverings	Monitor performance and gradually increase workload

Modernization Strategies

According to the Oxford Dictionary, "legacy" refers to something left or handed down by a predecessor. Legacy code is exactly this: code inherited from predecessors. More importantly, legacy code persists and turns into a durable and hard-to-extirpate entity because it just works. The more business depends on it, the more it can become a real threat. Any modernization effort begins with the best intentions to enhance technology, infrastructure and processes but it inevitably carries a high risk of hindering business continuity. Therefore, the challenge is discontinuing existing legacy software without disrupting the vital business continuity.

A more in-depth discussion of specific modernization strategies will follow in the upcoming chapter titled "Software Fitness."

Sparse Examples of Modernization Failures

Any software modernization effort is always undertaken with the noble aim of enhancing efficiency, improving user experience, and keeping up with technological advancements. However, not all modernization efforts are successful.

Especially if the new project starts with overambition and turns out overengineered, or if breaking

changes are rolled out brutally to all users at the same time, failure is definitely an option. Not necessarily a certainty, sure, but also an event with a nonzero likelihood to happen.

The news are full of examples of companies that faced substantial difficulties with some modernization or rewrite project ultimately reverting back to the older systems they planned to replace. Here are a few resounding failures that took place in the industry within the last two decades.

The Mozilla Big Rewrite

In the early 2000s, Netscape decided to rewrite from scratch the codebase of their popular browser Navigator. The project, known as the "Mozilla Rewrite," faced numerous problems, including delays, technical challenges, and disagreements among developers.

The rewrite ended up taking much longer than initially estimated. While Mozilla Firefox eventually emerged as a great outcome, the tumultuous process led to a dramatic loss of market share and ultimately sealed the Netscape's fate.

Too Ambitious to Succeed Windows XP

Longhorn was the codename for Microsoft's ambitious project to develop the successor to Windows XP, which eventually became Windows Vista in 2007. Originally envisioned as a major overhaul of the Windows operating system, Longhorn aimed to introduce revolutionary features such as a new file system (WinFS), a revamped user interface (Aero), and improved security and performance. However, as development progressed, Microsoft faced difficulties in implementing these features within the desired timeframe. As a result, many of the ambitious features, including WinFS, were either scaled back or postponed to future releases.

Following its release, Windows Vista received mixed reviews and faced significant criticism due to its performance, compatibility issues, and high system requirements. In 2009 Microsoft replaced Vista with Windows 7 which was well-received by users and became one of Microsoft's most popular operating systems.

When One Is Not Enough

Target Corporation is a major American retailer known for its discount department stores and hypermarkets. It ranks as the seventh-largest retailer in the United States and is listed on the S&P 500 Index.

Target Corporation attempted to expand into Australia in the early 2000s. The company implemented a new supply chain and inventory management system that was meant to improve efficiency and support its operations in the region. However, the system encountered significant problems, leading to inventory shortages, stockouts, and logistical challenges. Target Australia eventually scrapped the new system and reverted to its previous processes, resulting in significant financial losses and reputational damage.

In 2013, the company expanded into Canada with high hopes of success but faced again significant challenges with the inventory management system. Target attempted to rewrite the software to integrate their Canadian and U.S. systems, but the project encountered numerous technical issues and delays.

As a result, Target Canada struggled with inventory problems, leading to massive losses and the eventual closure of all Canadian stores in 2015. In its brief Canadian lifespan, the retail chain accumulated losses of $2.1 billion and the Canadian news media described the venture as "a gold standard case study in what retailers should not do when they enter a new market."

The Lidl's 7-years Case

In 2011, Lidl—a German global discount supermarket chain operating over 12,000 stores across Europe and the United States—reckoned the existing merchandise management and information system (Wawi) had reached its capacity limits and opted to replace it with a new solution based on SAP for Retail. The new solution aimed to integrate process chains from suppliers to customers, providing real-time key figure analysis and forecasts. Lidl also sought more efficient processes and simplified master data management for its 10,000+ stores and 140+ logistics centers.

In 2015, the new system went into production in Austria, Ireland, and the United States although with continued issues. In 2018, the board terminated the entire project because, looking at numbers of seven years of development, none of the originally defined strategic goals could be achieved with further reasonable effort. Not a decision against SAP, stated an internal memo, but still a necessary one. Lidl reverted back to Wawi and mobilized its IT team to make it future-proof.

The main reason for the failure, resulting in an overall write-off of approximately 500 million Euro, was the misalignment between Lidl's approach and the retail logic hardcoded in the SAP core system. The SAP system relied on retail prices for inventory, while Lidl's processes were based on purchase prices. Excessive customization and persistence in pursuing an unrealistic goal were key factors in the failure. However, perhaps more significant than the financial loss is the fact that the company spent seven years without modernizing or further developing its business processes.

There are a few common factors that could make the difference between failure and success. Overambition is one of them and also too low ambition is risky, when the team just attempts to convert existing code to a new stack. Existing code is likely a decade old—otherwise it would not be legacy code—and software technology evolves quite rapidly summing up at least three major releases in a decade. Just converting code might be too much work (and cost) for too little reward.

Employing cumbersome and buzzword-driven technology stacks is another looming obstacle on the way to successful modernization and insufficient agility—in the true sense of the original Agile Manifesto During deployment, a minor wrinkle may just amplify to become a problem in the first releases.

In a nutshell, the success of any modernization effort is subject to a variety of factors, and technological and architectural choices are just two of them. For example, what would be the ideal time to call for a modernization project within a company?

CHAPTER 3

Software Fitness

> *In programming, as in life, it's often the case that the best solution is the one you can get to work.*

Peter Norvig

Distinguished Education Fellow at the Stanford Institute for Human-Centered AI

A quick online search may reveal that the expression "software fitness" is sometimes used to refer to a set of tools and practices for assessing software quality and measuring how well it matches various levels of assigned specifications. In this context, however, software fitness refers to a set of techniques to get a given software application back in shape and enable it to meet its same (or extended and revamped) obligations in a more modern context, aligning with current regulatory and technological standards.
Beyond vital decisions about technologies, the first point to assess is when it is a good time to seriously plan the modernization of a legacy software application.

The Ideal Timing for Modernization

Crucial for maintaining competitiveness and securing the future of the business, modernization should occur neither too early nor too late, ideally aligning with significant advancements in technology or the business domain. Generally, there are several indicators that suggest time is running out.

- High maintenance costs and poor performance
- Demonstrated security vulnerabilities and compliance risks
- Use of outdated technologies
- Critical business inefficiencies and limited scalability
- Issues integrating the existing software with external platforms
- Growing user frustration

It goes without saying that each indicator must be supported and validated by concrete data and not simply by gut feelings. (Even though gut feelings do count!)

There are two dimensions to evaluate the feasibility of a modernization project: the business dimension and the technological dimension.

The Business Dimension

Are there security vulnerabilities?
Great to know, but which vulnerabilities in particular and how should they be addressed? Is there room for effective fixes and patches before rearchitecting? Can the application survive decently even with such vulnerabilities? What's the worst-case scenario?

Is a given technology currently in use outdated?
Sure, it may be, but what would be a viable alternative and how would implementing this new technology affect the existing codebase and hardware infrastructure? What would be the actual learning curve for the team to ramp up on a newer, or just different, framework?

Is performance somehow poor?
Indeed, it is. However, how is this underperformance affecting business margins or impeding potential expansion? Furthermore, which business processes suffer the most from the lack of performance?

This list could go on and on.

At the business level, any decision is a matter of numbers being above or below an acceptable

threshold. However, to help determine a realistic threshold, let's look at an age-based evaluation approach.

By rule of thumb, five years may be as long as a geological era. Assuming life on earth dates back to three billion years ago and software to only 50 years ago, it turns out that one software-year is the same as 60 million years for life. And in the geologic time scale, 60 million years is certainly within the range of an era.

Five years is a significant amount of time for a line-of-business application, especially considering the frequent changes in software technology that occur every few years. Hence, as the fifth year of a software's life approaches, the first alarm bell rings. You can hit snooze, but sooner or later, action must be taken.

The Technological Dimension

It is quite unlikely that the initial drive for a major rewrite would originate from the executive level. More likely, it is the architecture team that would recognize the need and initiate the call for such an undertaking, given their closer involvement with the technical aspects and day-to-day performance of the software.

But also in this case numbers should be the primary driver, not ignoring at all gut feelings. So, when to call out for software modernization? Here are a few parameters to consider.

Developer's Ramp-up Time

Ramp-up time refers to the period required for a developer to become fully productive and effective within a project and includes aspects such as understanding the codebase, getting acquainted with tools and technologies, grasping project requirements, adapting to the team dynamics.
Overall, the ramp-up time measures the learning curve associated with bringing a new developer into a new development environment.
How long does it take for a new developer to start contributing effectively to the codebase and placing acceptable pull requests? The ramp-up time depends on the developer's prior experience, and the quality of onboarding processes in place. The following table shows critical threshold values based on experience.

Numbers are just numbers and may realistically vary with the complexity of the project. Set proper numbers, though, a too high ramp-up time lowers productivity on maintenance tasks.

Level	Acceptable time to first commit
Newbies fresh from university	Less than two months
Mid-level (i.e., 1-3 years)	Three weeks
Experienced (3-5 years)	One week
Senior (> 5 years)	Two days

DevOps Facilities

DevOps enhances the software development process by increasing reliability while promoting collaboration and deployment efficiency of software applications. In the 2020s nobody questions the importance of DevOps in any software factory. Yet, DevOps practices are a relatively recent discovery that were just in the most intimate dreams of developers active in the early 2000s.

DevOps as a formal movement began around the late 2000s, with its foundational principles taking shape through early conferences and publications. Since then, DevOps has evolved and integrated with various other technologies and practices to become a cornerstone of modern software development and IT operations.

Is a project not suitable for DevOps concepts still worth the cost?
As weird as it may sound, there are a few aspects of legacy applications that may seriously hinder any attempt to automate deployment of fixes and updates. The trickiest aspect is interdependencies. Complex systems with intricate dependencies between services or components may require manual intervention to resolve conflicts or ensure correct sequencing. Different environments (development, testing, staging, production) may have varying configurations and requirements that can complicate automation. Differences in infrastructure setups across environments might necessitate manual adjustments.

In summary, not being able to automate deployment of fixes and updates is time-consuming and frustrating.

Completion Time of Operations

Related to the difficulties of automating deployment are the longer times to fix even simple bugs or to roll out new features. Both scenarios compromise competitiveness and raise frustration in both developers and end users.

On a personal note, recently it took me four hours to change a label—repeat, change a label—on a web page. It was due to the extremely complex routes that the composition of the page was going through. In the specific case, all the visual settings for the page were loaded from a common JavaScript repository created on the server and injected in the page and then pushed to a JavaScript event bus that each dynamic component was listening to. In the end, it looked like a task as simple as editing a resource string but turned out into a (long) nightmare. Worse yet, it's problematic to justify this to a manager or the stakeholder.

Rolling out a new feature is even a more painful scenario.

It affects the quality of the product if you don't do it; but if you do it you have to accept the risk that you're breaking something else.

Old-fashioned User Interface

Inevitably, every application is a product of its time and reflects the practices and customs of its era. In the case of software, the typical user interface often mirrors the design trends and usability principles of the time including user expectations and interaction patterns, color schemes, language, and iconography.

In the past two decades, there have been several significant changes in UI/UX design that may not be entirely reflected in legacy software. Here are some of the most impactful shifts:

- Mobile-first design
- Responsive design
- Minimalist and flat design
- Micro-interactions

Older applications tend to merge too many use-cases on too few screens resulting in visual noise that seemed acceptable in the late 1990s or early 2000s but it's not considered good practice nowadays. In addition, cumbersome user interfaces generate nontrivial issues with users' training and new developers' ramp-up.

Steps to Modernization

Modernizing a legacy application involves transforming it to leverage current technologies, improve performance, and enhance user experience while maintaining any core functionality. Any approach is articulated in three main phases:

- Assessment and Planning
- Selection of a Modernization Strategy
- Selection of a Rollout Strategy

Determining the goals of modernization is the initial step. Possible items in the list include improving performance, enhancing security, supporting new features, making the app cloud-compatible. The golden rule here is 'don't be over-ambitious' but try to stay focused on the existing working parts and identified pain points. Beyond programmatic statements, on a more practical note, there should be a priority list indicating which parts of the application need immediate attention and which can be addressed later.

There are various strategies for tackling modernization, which we will explore in detail shortly. Each strategy comes with its own cost/benefit balance and risk level for the management to consider. It ranges from moving the application to a new platform with minimal changes (e.g., migrating from on-premise infrastructure to cloud services) to progressive refactoring of the existing codebase up to a complete overhaul.

The call for a big rewrite typically comes from the architecture team driven by one or more of the above technology-oriented points. The final decision belongs to management and is typically inspired also by a psychological perspective. By calling a redesign, the management pushes the message that the team is catching up and fixing things quickly. The distance to walk to the next safe stand is constantly short and the finish line is always in sight.

By calling a rewrite, the management publicly embarks on an ambitious project, which is a challenge in itself. It has nonzero chances of success and nonzero chances of failure. It's like closing the eyes and start stabbing in the dark of hoped future greatness. By calling a big rewrite, the management seems to implicitly accept failure as an option.

Whether replatformed, refactored or completely rewritten, once the new application is ready, moving

it into production is the most delicate step. Ideally, users should be migrated to the new platform in small chunks to minimize disruption and manage risk with more comfort. During this phase, any further adjustments that proves necessary can be done with the least possible damage.

Strategy #1: Lift-and-Shift

Historically, the term lift-and-shift refers to moving an application from its current environment to a new platform with minimal changes to its architecture and codebase. This strategy is employed when organizations need to transition from outdated infrastructure to more modern platforms, such as migrating an application from on-premises servers to cloud-based environments. The primary advantage of lift-and-shift is speed and simplicity, as it allows organizations to quickly gain the benefits of the new infrastructure—increased scalability and reliability—while minimizing disruption.

However, while lift-and-shift offers a faster path to modernization, it typically does not fully address underlying inefficiencies or technical debt inherent to the legacy system. In the end, it is effective for immediate modernization needs, but should be followed by further optimization efforts, such as refactoring or re-architecting, to fully leverage the advantages of the new environment and more modern technology stacks.

Today, lift-and-shift tends to be used more as an umbrella term for two more specific terms rather than as a specific migration strategy. Lift-and-shift is commonly implemented through one of the following approaches:

- Re-hosting
- Replatforming

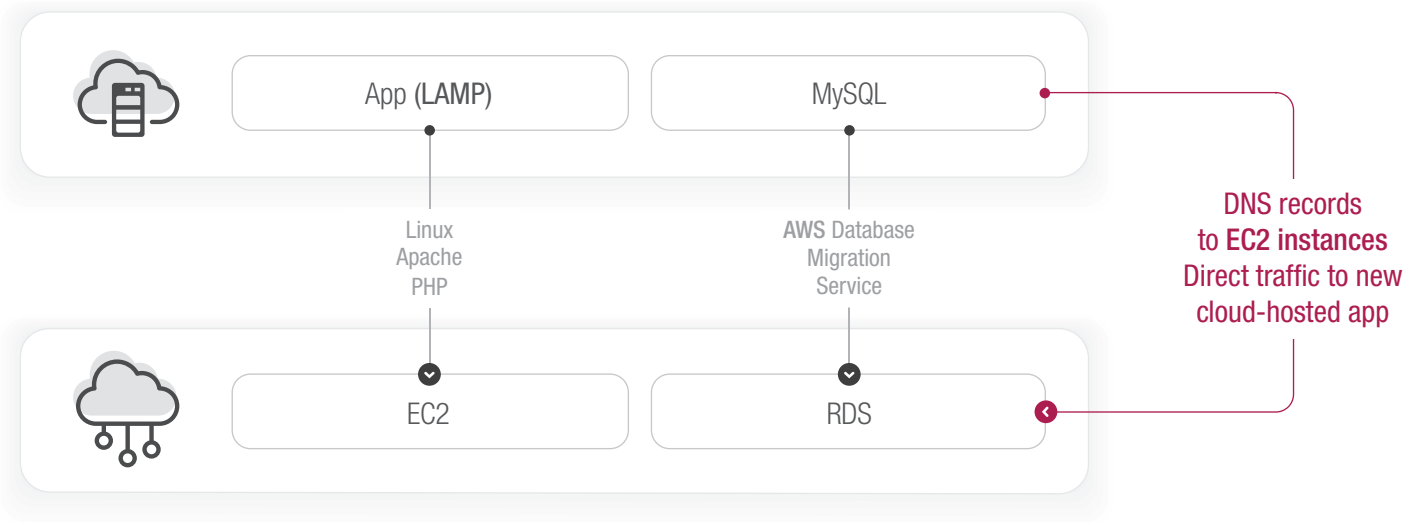
Both have their place as the initial step of a broader modernization strategy. The choice between re-hosting and replatforming depends on factors such as time constraints, budget, desired outcomes, and the state of the existing application.

Re-Hosting in Action

Re-hosting is essentially a new name for the original lift-and-shift strategy and, involves deploying the application in a new environment, such as a cloud platform, without significant modifications. The focus of re-hosting is on replicating the existing environment as closely as possible and as cheaply as possible. It is ideal for organizations that need a quick migration with minimal changes at low cost but it has important downsides. In particular, all existing inefficiencies, outdated technologies, and technical debt are carried over and none of the features of the new host environment (such as cloud-native capabilities) are leveraged.

An Example of Re-hosting

The figure below presents a canonical example of re-hosting an existing application. In particular, a plain web application based on the LAMP stack is moved from an on-premise server to Amazon Web Services (AWS).



The LAMP stack is a popular set of open-source software used to create dynamic websites and web applications. LAMP is an acronym that derives from the initial of four key technologies: Linux (the host operating system), Apache (the web server), MySQL (the database management system) and PHP (the central programming language).

Here are the necessary in the example:

- The files of the original application are transferred from the physical hard-disk of an on-premise server to a virtual machine space in an Amazon EC2 instance configured to run the LAMP stack.
- The local Apache configuration is adapted to point to the application's document root.
- The original MySQL database is exported to a file and the file is uploaded to Amazon RDS (Relational Database Service)—a managed database service that allows to set up, operate, and scale a MySQL database in the cloud.
- The DNS record of the original application domain is updated to point to the new EC2 instance. This can be done using the AWS Route 53 service or your domain registrar of choice. In this way, any traffic to the assigned domain is now redirected to the EC2 instance.

The new application is now hosted by a EC2 instance—a web service provided by AWS that offers resizable compute capacity in the cloud.
An analogous re-hosting could be done targeting the Microsoft Azure cloud platform. The steps to take are the same; the involved services are functionally analogous. You host the new application in an Azure Virtual Machine equipped with your favorite Linux distribution and import the original MySQL database in a new Azure database service for MySQL. This approach offloads database management tasks to Azure and provides automated backups, scaling, and high availability. In alternative, you can manage MySQL directly in the Azure virtual machine in much the same way you would do in the on-premise environment.

EC2 instances and Azure virtual machines work similarly. Both run on physical servers and share those physical resources through virtualization technology. Both provide isolated environments for running applications and both can run various operating systems (e.g., Linux, Windows). The choice between them may depend on specific business needs, existing technology stacks, and integration requirements. AWS EC2 is often preferred for its extensive service offerings, while Azure virtual machines are favored for the seamless integration with Microsoft products.

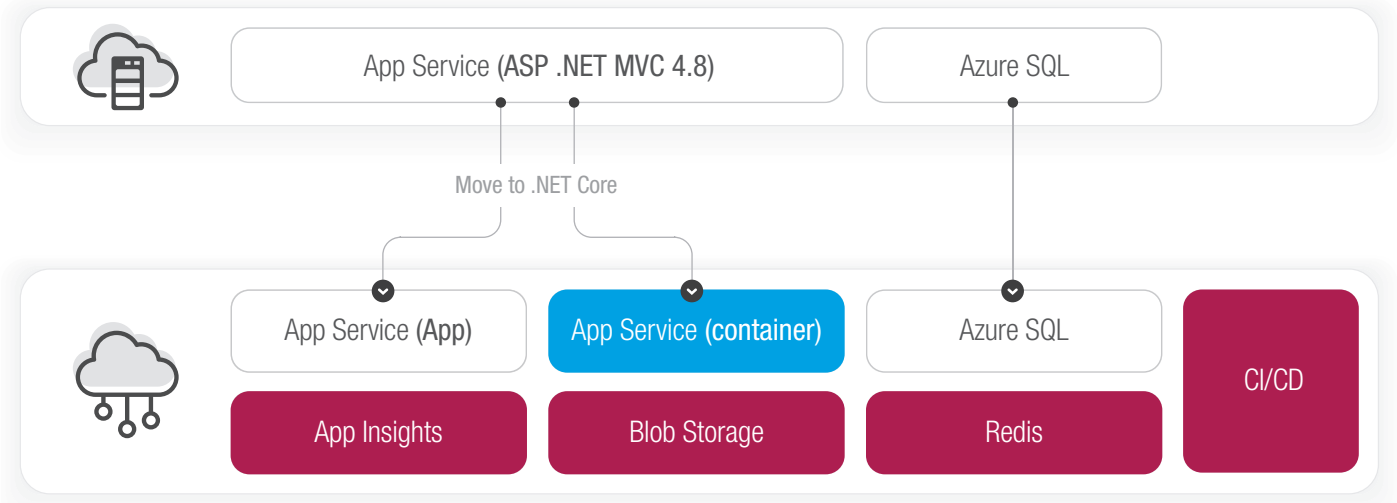
Replatforming in Action

Replatforming is a migration strategy where you move an application from one platform to another with minimal changes to the codebase. Replatforming is move intrusive than re-hosting in that it assumes changes and fixes at various levels of code and infrastructure to improve performance, take advantage of cloud-native features, or optimize costs. However, unlike a complete rewrite, which involves redesigning the application from scratch, replatforming aims to transition the application to a new environment and/or technology stack while retaining its original functionality.

The foundation of replatforming is to make as few changes as possible to the application's code. The primary focus is on adapting the application to the new platform without a complete overhaul. For example, you might move from an outdated technology stack to a more modern one, such as upgrading from an old content management system (e.g., Joomla) to a newer version or an alternative (e.g., Wordpress).

An Example of Replatforming

The figure below presents a first example of replatforming a legacy ASP.NET application. The original application is already deployed to Azure and uses an Azure SQL database service. The source code is based on the ASP.NET MVC 4.8 framework.



ASP.NET MVC 4.8 is considered a legacy technology because it is based on the older .NET Framework, which is now in maintenance mode. This means it continues to receive security updates, but it will no longer receive feature updates or major enhancements. In the last decade, in fact, the industry has been moving towards .NET Core, which offers modern features, cross-platform capabilities, and relevant performance improvements.

Upgrading an existing ASP.NET MVC 4.8 applications is challenging because it inevitably requires the migration to .NET Core. Key differences lie in the following areas:

- Sparse API changes in the .NET Framework
- New configuration system based on settings files
- Built-in dependency injection layer
- Startup and middleware configuration
- Authentication and authorization
- Routing and URL management
- Sparse changes in views and Razor syntax

Migrating to a .NET Core application may be a simpler matter of fixing each of the points above individually, then coding and tinkering to a running version of the code. Or it can be a matter of learning enough about .NET Core and the structure of web applications and then building a new application from scratch.

More work can be done, though, that marks a key difference with the re-hosting approach. For example, some of the functions can be exported to a separate (possibly dockerized) component. More cross-cutting services can be leveraged from the cloud environment: AppInsights for performance monitoring, blob storage for storing binary resources, Redis for cross-server caching and DevOps services for managing code repositories and creating releases.

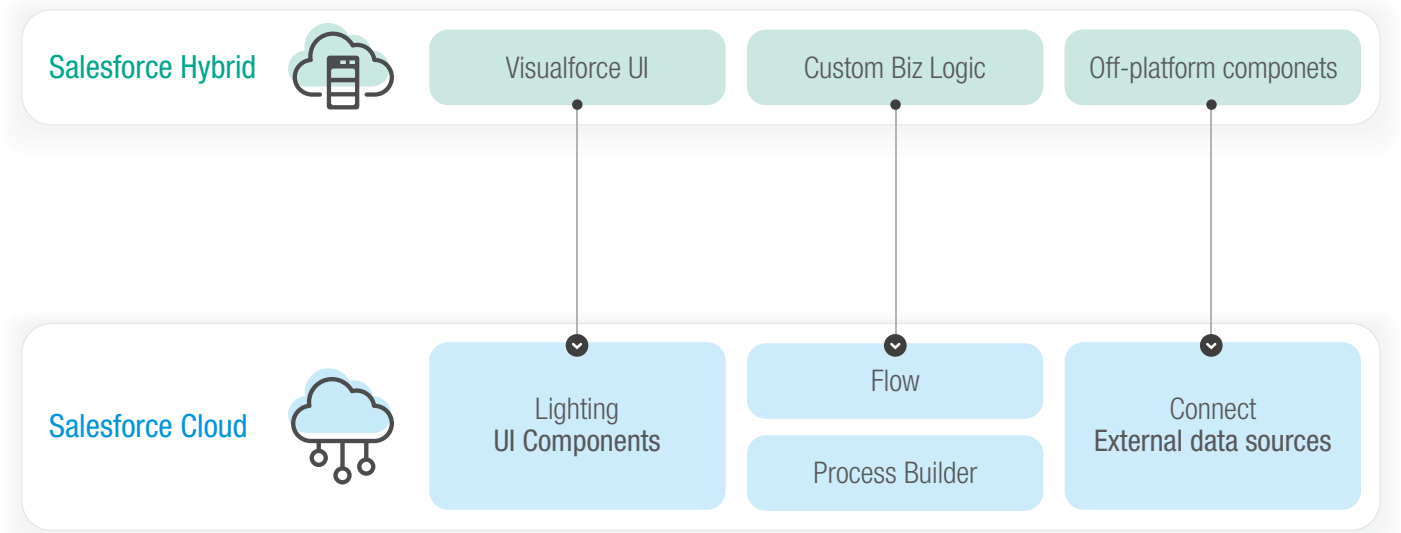
The same legacy application is now hosted in a newer environment but, at the same time, it uses a newer runtime, has the source code partly refreshed and integrates with available services for a richer business ecosystem.

Another Example of Replatforming

In the previous example, we assumed the migration of the source code from ASP.NET MVC to .NET Core. The distance between the two platforms—no matter the availability of tutorials and rich documentation—is not minimal. Frankly, migrating to .NET Core is often halfway between replatforming and rewriting. Here's, instead, a more canonical example of replatforming.

The starting point is a custom Salesforce application with extensive custom code and some off-platform components. It may be hosted on-premises or in a hybrid environment. The final destination is an application deployed to the Salesforce Cloud leveraging Lightning Components, Salesforce Flow, and maybe other platform-specific services.

The first step in the migration path consists in converting the custom code in the Visualforce pages to Lightning Components to enhance user experience. Controls in the frontend trigger workflows. In the new platform, you replace custom workflows or business



logic with Salesforce Flow or Process Builder automatable processes composed with a visual designer. In addition, you may use also use server-side scripts written in the Apex programming language that execute in response to specific changes in Salesforce data. Finally, you employ Salesforce's native integrations (e.g., Salesforce Connect, External Services) to replace custom-built integrations.

As in the picture, any module in the original application has a direct counterpart in a newer version of the Salesforce environment. Migrating is a sort of point-by-point translation and mapping from a set of tools to another. Both Visualforce and Lightning Components serve important roles in Salesforce development. Visualforce is great for building straightforward, server-rendered pages with complex business logic. In contrast, Lightning Components offer a more dynamic and highly responsive framework for creating sophisticated user interfaces that align with contemporary web development practices. Furthermore, Salesforce Flow is ideal for creating both interactive and automated complex workflows, while Process Builder is great for simpler, straightforward automation tasks. Both tools help automate business processes within Salesforce, reducing the need for manual intervention and increasing efficiency.

In the whole replatforming effort the time spent in rethinking the application and its use-cases is minimal– the existing core functions have been ported to a new runtime environment and adapted to work effectively.

Strategy #2: The Big Rewrite

At the very end of the day, beyond re-hosting and replatforming, everything else is, by all means, a "big rewrite". What is a "big rewrite", exactly?

A big rewrite is when the entire application is redeveloped. This often means discarding the old codebase and building a new one, potentially using different

technologies, frameworks, or architectures. A big rewrite provides an opportunity to redesign the system's architecture, user interface, and user experience from the ground up. During a big rewrite, the team typically reassesses the application's requirements to ensure they align with current business needs and technological advancements.

To developers, the theme of the big rewrite has the same dark fascination of moths to flames. It is intriguing to plan, but it is also frustrating because it often leads to stress and poses risks to the business. As mentioned earlier, though, there are some dry numbers to measure the actual need for a rewriting project. If a simpler lift-tinker-and-shift approach doesn't just work, everything else has the haunting allure of the big rewrite.
Only one point remains to assess: should it be an incremental redesign or a brutal rewrite?

Incremental Redesign

The notion of incremental redesign is quite easy to grab, yet it is not so obvious to adopt. Here's a widely accepted definition: Incremental redesign of software involves making gradual improvements to a software application rather than undertaking a complete overhaul. It focuses on evolving the application through a series of small, manageable changes, allowing for continuous refinement and adaptation. Here are a few cross-cutting facts about incremental redesign.

Codebase. You work by creating new branches on the same repository. This means that no new greenfield project will be created. It also means that large chunks of the code will remain untouched. At best, critical code will be "sandboxed" in safe blocks and commanded through some new layer of API. New branches will focus on any refactoring necessary for implementing new functions in full safety and on actual coding of new functions.

Technical Debt. Nearly no effort will be made to refactor and modernize the codebase. Quite likely, no modern language features will be leveraged to make the code more readable. In these conditions, keeping the technical debt constant across the redesign project would indeed be a huge success. Experience shows that injecting hours of work for mere technical debt reduction and increase of readability is technically possible but left to the nerve and audacity of individual technical leads.

Time-to-Market. The resulting time-to-market of an incremental redesign project is always the fastest possible compared to a major overhaul of the code. Releasing incremental updates is typically quicker than waiting for a comprehensive redesign. Furthermore, frequent updates enable smoother integration of user feedback minimizing the likelihood of introducing major bugs or keeping the downtime to a minimum.

How large is an incremental redesign project? Put another way, how many incremental steps are expected for such a project? The size and the number of such increments is a crucial factor in the general outcome of the project. It may mark the difference between general satisfaction and negative feedback and between budgetary wins or losses.

A few incremental changes amount to mere maintenance rather than a true redesign. Conversely, when

many incremental changes are made, the project can end up in a full rewrite just done in stages. Keep in mind that, in such cases, the overall cost of incremental redesign is generally higher than that of a complete rewrite. However, the impact on the organization is typically less severe. Put down in terms of cost effectiveness, whether incremental is preferable over the big overhaul is not obvious. The appeal of an incremental solution decreases with the number of planned increments.

Things One Commonly Expects from a Rewrite

Whether incremental or full, a rewrite brings a number of general-purpose expectations. In the end, the new revamped application should be better in a number of aspects. On the technical side, this may require any of the following:

- Expand one or more monolithic blocks
- Full replacement of core business or infrastructural modules
- Add brand new functions across existing monoliths
- Modernize the user interface

A legacy application is typically made of monolithic blocks that hard-code a crucial business function. The block is solid and works great. However, it must be extended with new capabilities. The approach is just one: secure and fortify the block drawing clearly and neatly its perimeter. Then, act inside of it with the absolute certainty of not breaking anything outside.

Full replacement of a core business module or an infrastructural layer (e.g., authentication, caching) requires a similar strategy: identify the perimeter and secure it from possible leaks of references and function calls. Furthermore, it would be desirable to abstract the public interface of the module before replacing it with another module, completely rewritten and autonomous, that exposes the same interface and because of that magically plugs into the existing outlet.

Adding brand new business functions is probably the hardest scenario of an incremental redesign project because no general rules may exist. Effort and difficulty depend on the specific case. It is realistic to expect that a new function runs across multiple existing blocks forcing the team to put hands in many different places. Sandboxing each block beforehand may be quite expensive and raises the overall effort up to that of a full rewrite without reducing the technical debt—fundamental outcome of a serious big rewrite. On the other hand, having no preliminary sandboxing may expose the unexpected: hidden and undetected dependencies that pop up suddenly, delaying the effort and raising concerns.

Compared to business concerns, how can one be concerned about the modernization of the user interface? Well, simply upgrading a large CSS framework to a new version may be painful. Changing the sole CSS framework is challenging. If you also want to change the rendering engine and underlying logic (e.g., React, Vue, Angular, ASP.NET, Svelte, etc.) then the sole rebuilding of the frontend turns into a major overhaul.

Is Incremental Always an Option?

An incremental redesign is always a worthwhile option to consider, although its appeal diminishes with the number of planned "increments" and the actual amount of work required to apply each of such "increments". All this said, though, the major problem is that many existing applications don't just lend themselves to be incrementally redesigned due to technological and strategic factors.

Technological factors. The main obstacle is the degree of coupling between modules. Simply put, a high level of coupling makes each component potentially susceptible to changes made to others. As a result, even a minor and isolated change can trigger a series of necessary and cascading modifications throughout the entire application. Not to mention then the entire redesign of a component. In this case, in fact, you should be ready to face repercussions at least on all surrounding components.

Another impeding factor is the impact of planned changes to the technology stack. If it proves necessary to touch infrastructure and/or frameworks the final complexity would explode making the cost/effectiveness advantage of incremental updates vanish.

Strategic factors. Depending on the number of planned "increments", a redesign project may not be a short-term project at all. This means that stakeholders will see a steady expenditure of the budget for a goal that seems elusive—ultimately, remains to be the same application! Not that with a big rewrite project stakeholder won't complain for the continuous budget drain. In this case, though, they have a stronger reason to continue to avoid sitting in the nowhere land between a half-done new application and a half obsolete existing application. In this situation, however, they have a stronger incentive to continue to avoid being stuck in the limbo between a partially completed new application and a partially outdated existing one.

Everyone has their own attitudes towards risk and their own bottom-up or top-down perspectives on building things. Faced with the same need for expansion and modernization, some will be always more inclined towards a complete rewrite, while others will always prefer incremental extensions. I personally tend to identify more with the big-rewrite camp. However, my strong attraction for greenfield projects takes me to commit to rewrites very often—any five years old software is a good candidate for a rewrite. In the end, the gap between the technological snapshot of the current time and five years before is relatively small so that the difference between an incremental redesign and a big rewrite tends to blur.

Before You Jump on a Big Rewrite

Great! So, welcome to the exciting new big rewrite project. Here are a few key prerequisites to be aware of:

- Without the strategic commitment of executives and the support of end users, failure is an option.
- Without influential figures (e.g., chief architects, executives, stakeholders) investing their reputation in the project's success, failure is an option.
- Without a few top-tier developers, failure is an option.

Before you embark on a rewriting project, have a thought about some tips from those who have been there and came back safe and sound to tell their stories.

Greenfield coding. Do not try to convert existing code but design the new application from scratch. Conversion is only apparently a shortcut and time-saving approach. You may save some time to the first alpha release but, in the long run (i.e., the final release candidate) it won't be faster and you won't recover much of the technical debt in the folds of the old code and design philosophy.

Technology stack. Keep it minimal and as streamlined as possible. Avoid the temptation to over-engineer. Focus on doing things properly with only what is necessary—nothing more, nothing less. The Romans built roads and structures that have lasted for over two millennia without intending for them to do so; they achieved this through extreme rationality and without unnecessary embellishments. You should adopt the same approach. Add only the technologies and libraries that are absolutely essential, and scrutinize every new addition, even a small CSS file. Develop internal libraries and frameworks as needed, and avoid spending months on a framework that theoretically does everything but has never been tested with real code in real use-cases.

User involvement. Consider involving end users directly in the design process, but don't allow them to dictate how it should be done. The end user opinion matters as long as the process steps and overall experience are concerned and never at the cost of sacrificing sound design principles. The feedback of users is invaluable only if merged with the architect's design.

Rollout strategy. Avoid opening the new system to the entire audience all at a time. Instead, manage to migrate users to the new system team by team ideally starting with the teams that experienced the most issues with the current version.

Finally, two more outcomes right from the trenches of completed big rewrites:
First, during the rewrite you may discover that the logic you were scared to touch is not scary at all. Many bugs in the old system were due to its inherent design flaws rather than the actual logic of how the ideal system should function.
Second, reading code is usually much harder for the average developer than writing new code but without reading the same code you are replacing you may lose the knowledge accumulated in the old code. And this kind of domain knowledge is project heritage to preserve.

Flavors of Big Rewrite

No matter what, the big rewrite sometimes happens. The first law of big rewrite is that no success story is really repeatable and every project story, whether successful or failing, is unique.
Nowadays, major rewrites occur to eliminate outdated, decades-old legacy code or to meet the hyped demand for infinite scalability. A successful strategy to secure funding is to draw a parallel between your company and tech giants like Netflix, Amazon, or Uber, which operate massively distributed architectures.

Out of this, I recognize two flavors of big rewrites: buzz-driven and isola-et-impera. The former is based on a kind of tech ideology; the latter is more driven by sane pragmatism.

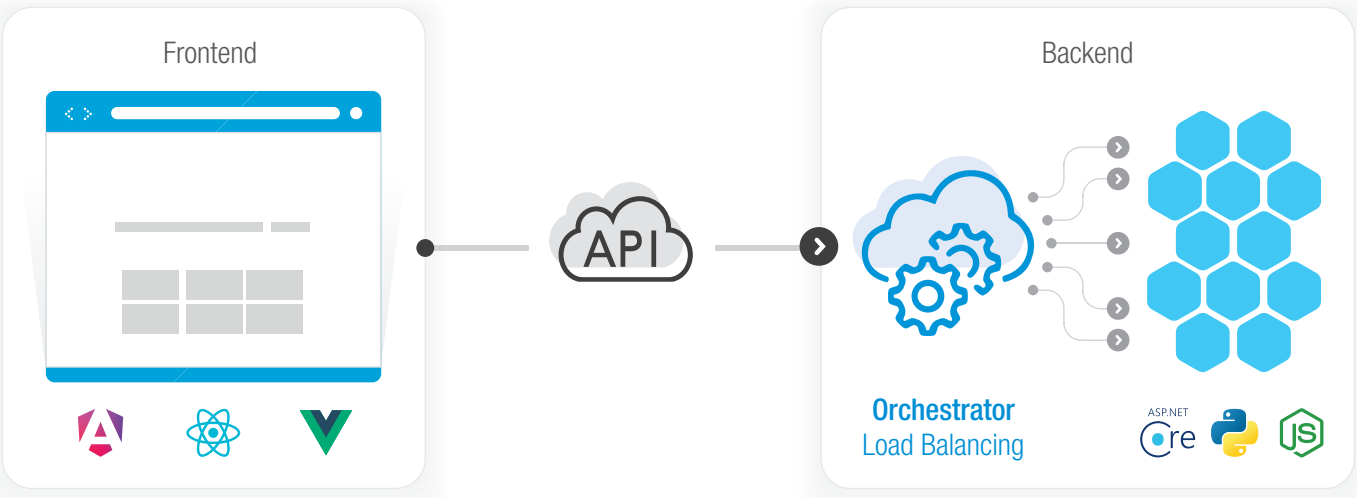
Buzz-driven Rewrite

Companies that lately embarked on big rewrite projects ended up with fairly convoluted architectures mainly centered on mainstream frontend technologies like Angular or React to give users the thrill of a modern user experience. On top of the frontend, some GraphQL query layer picks up data from a variety of heterogeneous resolvers and mutators working on top of the existing—legacy—code and underlying databases. The business logic gets wrapped up in GraphQL-friendly minimal components and parts of the legacy application that can't be easily modularized survive in Docker containers and communicate in some way with the rest of the world. This is a schema that easily labeled as "microservices". The figure below summarizes a view of the architecture from a fairly high level of abstraction.

An orchestrator in microservices architecture is essential for automating the deployment, scaling, monitoring, and management of microservices, ensuring that they operate reliably and efficiently within the distributed system. Tools like Kubernetes, Docker Swarm, and AWS ECS are popular choices for an orchestrator that provide comprehensive orchestration capabilities.

All microservices fit together like pieces of a puzzle, working together to form the complete system. This approach allows for greater flexibility, scalability, and resilience, as well as easier maintenance and updates compared to monolithic architectures.

While microservices offer numerous benefits, they also introduce considerable complexity in terms of implementation of cross-cutting concerns (logging, caching, authentication), data and transaction management, and operational overhead. Each microservice is an autonomous component of possibly any size, ranging from a stateless function to an entire functional module.



Moving a large legacy application to microservices goes against two big hurdles:

- The inherent difficulty of designing and implementing a microservice architecture from scratch (which is different from what tech giants actually did—they arrived to a microservices distributed architecture driven by the need of serving humongous number of requests.
- The domain-specific difficulty of breaking existing legacy functions into well-partitioned, function-savvy, manageable, autonomous and isolated services.

Adopting a microservices architectural mindset for new development typically involves three primary risks:

- Creating a mess of poorly designed endpoints
- Composing a thick layer of additional frameworks and services to give the messy collection of endpoints some clear functional sense
- To avoid this, or to remedy, falling into the trap of big upfront design

The first risk arises from the hazardous combination of unclear objectives and the pressure to deliver quickly to the market. To prevent the creation of a chaotic and unwieldy web of minimal endpoints, which can be difficult to safely remove, the team must conduct a thorough initial assessment. This assessment is often based on assumptions about how the system might evolve in an unpredictable future. Such a lengthy initial evaluation closely mirrors the notorious big upfront design. Even more problematic, it may end up being a form of big upfront design carried out amidst highly dynamic and uncertain requirements.

Isola-et-Impera

The ultimate goal of software investment is to create a successful application, not just a successful microservices architecture. Therefore, as a software architect, your focus should be on ensuring the overall project's success rather than solely the microservices architecture.

Many microservices success stories follow a clear, staged progression, similar to the growth of a majestic tree. It begins with the establishment of a strong monolithic foundation, which serves as the system's roots. Over time, as the system grows, its size and complexity can become unwieldy. Like a grand tree that requires careful pruning, the original monolith is eventually divided into a network of microservices that can operate independently while maintaining harmony within the ecosystem.

The success story of a modern application typically unfolds in four basic stages:

- Building the initial monolith
- Expanding the monolith according to business needs
- After some time (which may span a few years), the monolith may become unwieldy
- If this happens, consider migrating to a more distributed architecture

In such cases, microservices can be a viable option for achieving a more distributed architecture.

Modular Monoliths

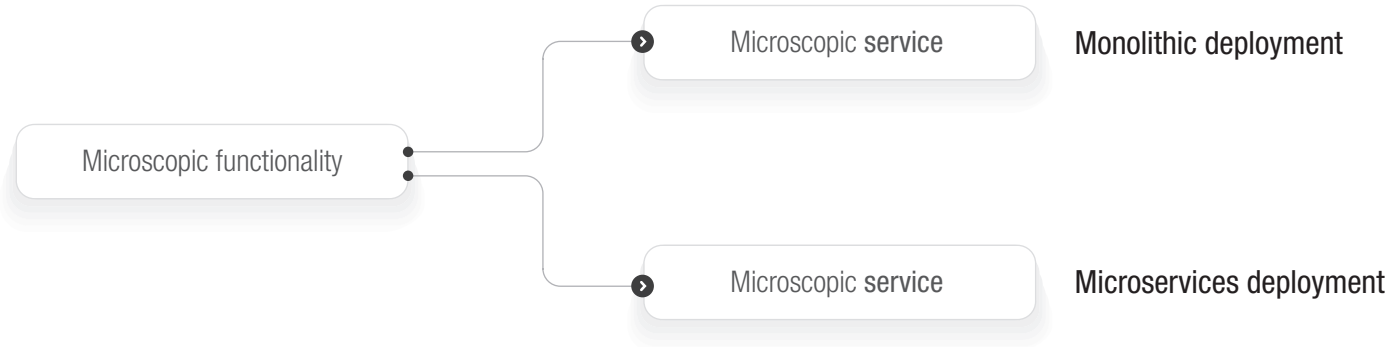
The word "monolith" is sometimes unfairly viewed as obsolete and antiquated. Such strong aversion is understandable when dealing with legacy monoliths that have a codebase riddled with hidden dependencies, that are untestable, poorly documented, and resistant to change. However, a newly developed monolith, designed from the ground up with the guiding principles of separation of concerns, embodies the concept of a modular monolith. And a modular monolith possesses several intriguing qualities.

While the entire application is constructed as a single codebase, it is organized into loosely coupled modules, each serving as a self-contained unit of functionality. An application developed this way is typically quick and easy to deploy, whether through a DevOps CI/CD pipeline or manually using a publishing profile. In addition, testing, debugging, and logging are straightforward in a monolith, and pose no additional issues as for microservices, just because everything is in one place.

Monoliths can offer superior performance because in-process communication is much more efficient than inter-service communication inevitably used in microservices. This minimizes latency and ensures data consistency by default.

On the downside, scaling a monolith can be challenging because, as a single deployable unit, you cannot selectively allocate resources to specific areas in need. You can enhance the underlying database's capacity (which often suffices) or deploy multiple instances of the application behind a load balancer.

However, once you have logically decomposed the business logic and processes into isolated components—whether newly written or legacy code sandboxed as standalone modules, you have a number of deployable services, as small as reasonably possible. If you bundle all these "microscopic" services together, you have a (highly modular) monolithic deployment. Otherwise, as the figure shows, you can partition the set of microscopic services into multiple deployable units orchestrated and managed as appropriate. In other words, choosing the monolithic way doesn't preclude future growth of scale and doesn't limit future scalability, should it become a strict necessity. At the same time, starting with a monolith saves burden and costs and significantly increases the chances of success of the whole rewrite project.



The evocative term “isola-et-impera” just aims at reaching the maximum possible level of modularity in the original design. With a successful logical decomposition of functions, you can go anywhere.

Cloud-nativity

Alongside microservices, another prevalent buzzword in tech news is cloud-native, which is often seen as a crucial requirement for new technology projects. Cloud-native applications are designed from the ground up to fully leverage cloud computing environments, aiming for modularity, scalability, and resilience. But what sets them apart from other modern applications?

In reality, there is no significant difference; “cloud-native” is simply a term used to describe modern software systems. However, the cloud-native label does not apply to applications that are merely migrated to the cloud through lift-and-shift methods, where the core application remains unchanged but is hosted in a cloud environment.

New applications are inherently cloud-native because their core is deployed in the cloud and they are designed to connect and utilize additional cloud services such as databases, blob storage, logging, caching, DevOps tools, and containerization.

A Pragmatic .NET Rewrite Approach

Let’s now focus on a specific technology scenario to explore legacy application modernization in more detail. The reference scenario is an outdated, years-old ASP.NET application on the canonical Microsoft .NET stack and the Azure cloud environment.

The .NET Core Factor

Doubtless, .NET has significantly impacted web development since its introduction in the early 2000s. One of the primary contributions of .NET to web development is its comprehensive and integrated environment. The introduction of ASP.NET provided developers with a cohesive set of tools, libraries, and languages, including C#, VB.NET, and F#. This integration simplified the development process, allowed developers to work within a unified ecosystem that supports the entire development lifecycle, from design and coding to testing and deployment.

A decade ago, there were two main ways of building web applications targeting the Microsoft stack: ASP.NET Web Forms and ASP.NET MVC. Both leveraged the same runtime engine devised in the early days. In other words, ASP.NET MVC was built as a parallel pipeline diverging from the main through dedicated middleware at a point of the chain. Later on, ASP.NET Web API was mounted on top of ASP.NET MVC as yet another parallel pipeline departing from yet another route point.
Around 2014, in the middle of a silent battle that could have changed the face of the web of ten years later, Microsoft opted for rebuilding .NET from the ground up and made it open-source and cross-platform since the first byte. That is .NET Core.

.NET Core was welcomed as a savvy way to bring the web back to its original tracks of HTML, CSS, and JavaScript getting rid of fancy things such as

Silverlight and C# loaded into the browser. Paradoxically, though, the codebase from which the whole .NET Core framework sprout is just the (cross-platform) codebase of Silverlight!

Today, choosing the Microsoft development stack means migrating any existing codebase to .NET Core at first.

Non-.NET Core applications become more and more outdated every day. To design a microservices architecture or a modular monolith, .NET Core is the ideal framework because it aligns with contemporary web development trends that emphasize agility and resilience. Additionally, .NET's support for containers and orchestration tools, such as Docker and Kubernetes, enables developers to create scalable and resilient web applications that can be easily managed and deployed in cloud environments.

Migrating the codebase to .NET Core is just a code-level form of lift-and-shift. More is needed to justify the investment and really take a more solid and strategic position for the next decade at least. Beyond schematic views such as incremental design or big rewrite, a wiser approach is looking for an effective way of teaching an old dog some new tricks

Teaching an Old ASP.NET Dog New Tricks

The phrase "you can't teach an old dog new tricks" is a popular English proverb that suggests it is difficult to make someone change their habits or learn new skills, especially if they have been doing things a certain way for a long time.

The phrase aptly captures the challenge of modernizing legacy software applications. Legacy systems have deeply embedded workflows and technologies, making them resistant to change, much like an old dog set in its ways. These systems are built on outdated technologies that are nontrivial to update, analogous to the entrenched habits of an old dog. Stakeholders and users familiar with old systems may resist change due to comfort with the status quo and fear of new learning curves. Modernizing requires putting resources on the table and it carries risks such as potential downtime and data loss, similar to the effort and uncertainty in teaching an old dog new behavior.

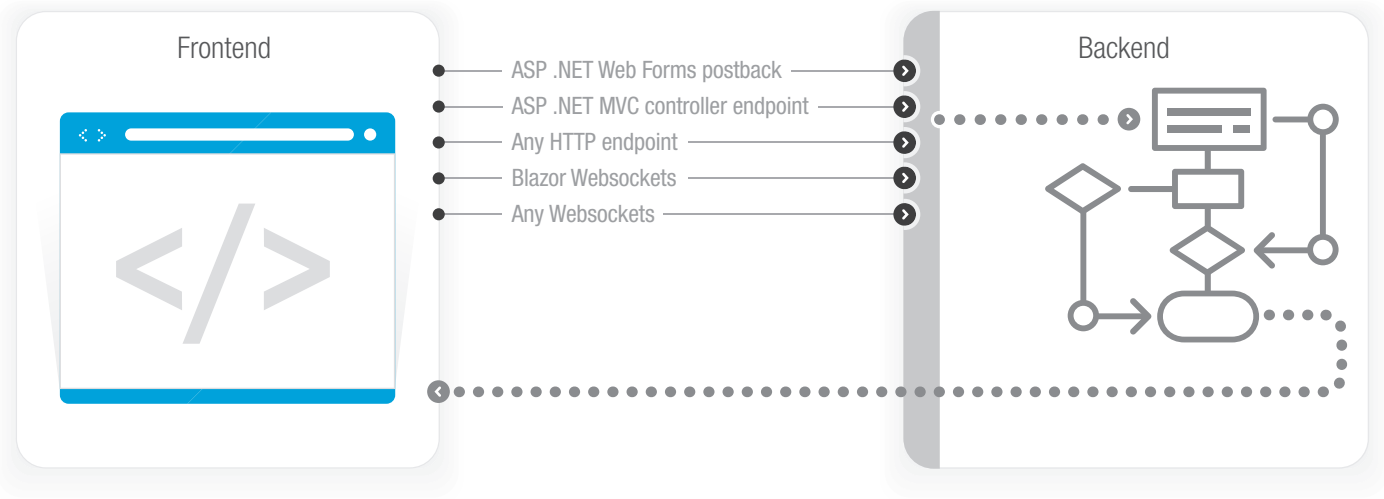
Despite these challenges, though, successful modernization leads to significant improvements, making it a worthwhile endeavor even for entrenched legacy systems. The challenge is finding that effective, and relatively smooth, way to re-train the old ASP.NET dog to play much needed new software tricks.

X-Raying the Inner Workings of Any Web Application

The figure below encapsulates the skeleton of nearly any ASP.NET application not already specifically designed for .NET Core and a modular architecture.

Any web application for the Microsoft stack has any of the following traits depending on actual technologies employed.

Frontend. In a classic ASP.NET (Web Forms or MVC) application, the frontend is made of server-generated HTML padded with manual or auto-generated JavaScript functions to post data and receive responses.



If the application is based on a Blazor (or even a Silverlight) frontend then the application logic is C# executed on the client. It could even be that the whole frontend (or just a part of it) has been replaced with standalone frontends based on more modern script-based frameworks using TypeScript or proprietary dialects of JavaScript.

Posting mechanism. The ultimate purpose of any web frontend is only one: invoking endpoints on some backend gateway. This can happen in a finite number of ways. The most common is a HTTP request sent to a custom (or RESTful) HTTP endpoint. In other cases, it can be a web socket endpoint backed by a SignalR, Blazor, or RabbitMQ server. Even less likely, it can be a call directed at a gRPC-connected backend.

Business logic. The backend of a web application encompasses operations like validating user inputs, managing data flow, executing core functionalities (e.g., calculating discounts, processing orders), and interacting with databases and other forms of storage. Depending on the thickness of the abstraction layer, triggers of business actions may be directly accessible to external endpoints or mediated by use-case orchestrators.

In a nutshell, any web application works by connecting the dots between client links and server endpoints.

Don't Touch My Backend

The golden rule of any refactoring effort–and modernization of a legacy application is probably the most complex form of refactoring–is "don't touch my backend". The slogan underscores the importance of respecting boundaries between frontend and backend development to maintain a clean, efficient, and collaborative workflow.

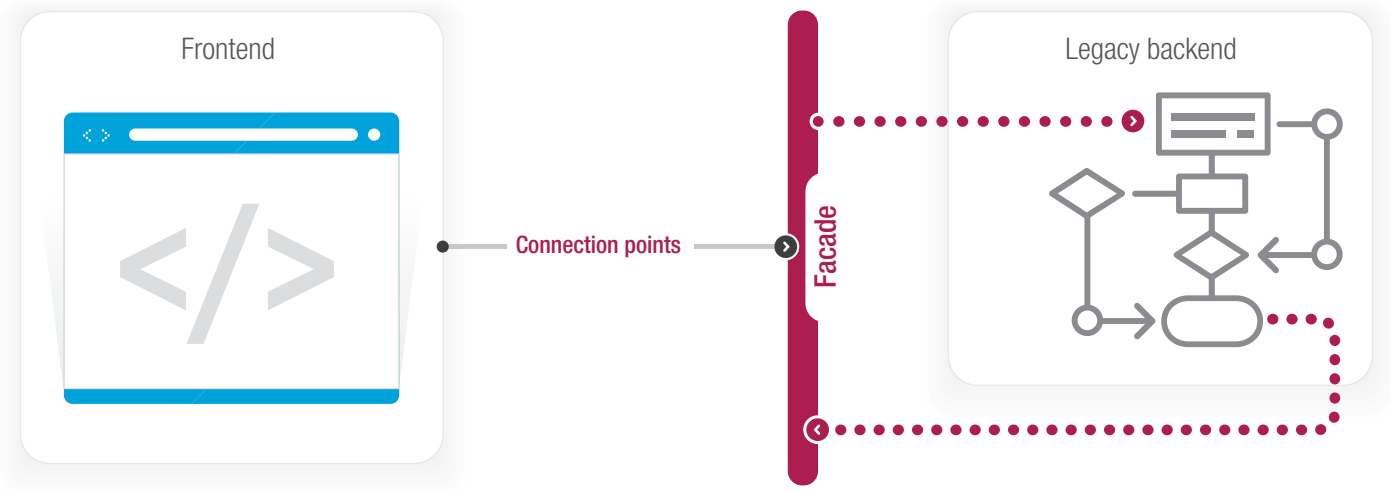
Furthermore, in a modernization perspective, "don't touch my backend" reminds the importance of not altering consolidated workflows that passed the test of time and ensure and back up one or more crucial lines of business. A Big Rewrite approach, per se, involves putting hands on every aspect of the old application and rewriting everything or merging new and old code. This is the biggest risk to face. Is there a

way to modernize the look-and-feel, architecture, and infrastructure of an existing application without risking breaking the backend and its embedded business logic? Ultimately, modernizing a legacy application is akin to recovering from a physical injury.

Imagine you suffer an ankle or knee injury. Any doctor would apply a rigid bandage to the ankle and not the whole leg. Any doctor would also recommend to walk with the aid of crutches or other supportive devices. What's the point of using crutches?

Crutches help offload weight from the injured leg, prevent further injury, and promote healing. Crutches also provide support and stability, allowing individuals to move around more easily despite the injury. This can help maintain independence and prevent the need for prolonged bed rest. Finally, crutches assist in the gradual reintroduction of weight-bearing activities during rehabilitation. As the injury heals, crutches allow for controlled, incremental increases in weight and activity levels.

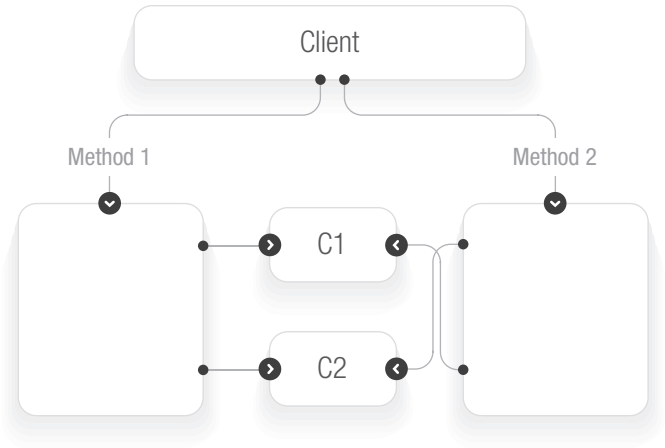
To protect the backend, you need to isolate and protect sore points. In a way, create the software counterpart of crutches that preserve the functionality of the backend while proceeding with the rehabilitation of the whole application. Isolating sore software points means designing an ad hoc facade over the actual business functions of the backend. The figure below builds from the previous one to show the point of having such a facade.



Building an Ad-hoc API Facade

When it comes to concrete implementation, the facade becomes a collection of endpoints to trigger selected and required business actions. After analyzing the inner structure of the legacy backend, the team should come up with a list of functions that can be triggered in isolation—take some parameters and return some results. These functions form the ultimate API facade.

It must be noted that not all identified endpoints may exist already in the legacy codebase. More often than not, such endpoints must be created or adapted to work in isolation. Most of the effort of modernization is just here—in breaking down the business logic into isolated blocks that can be deployed and invoked in total isolation. Here's an example.

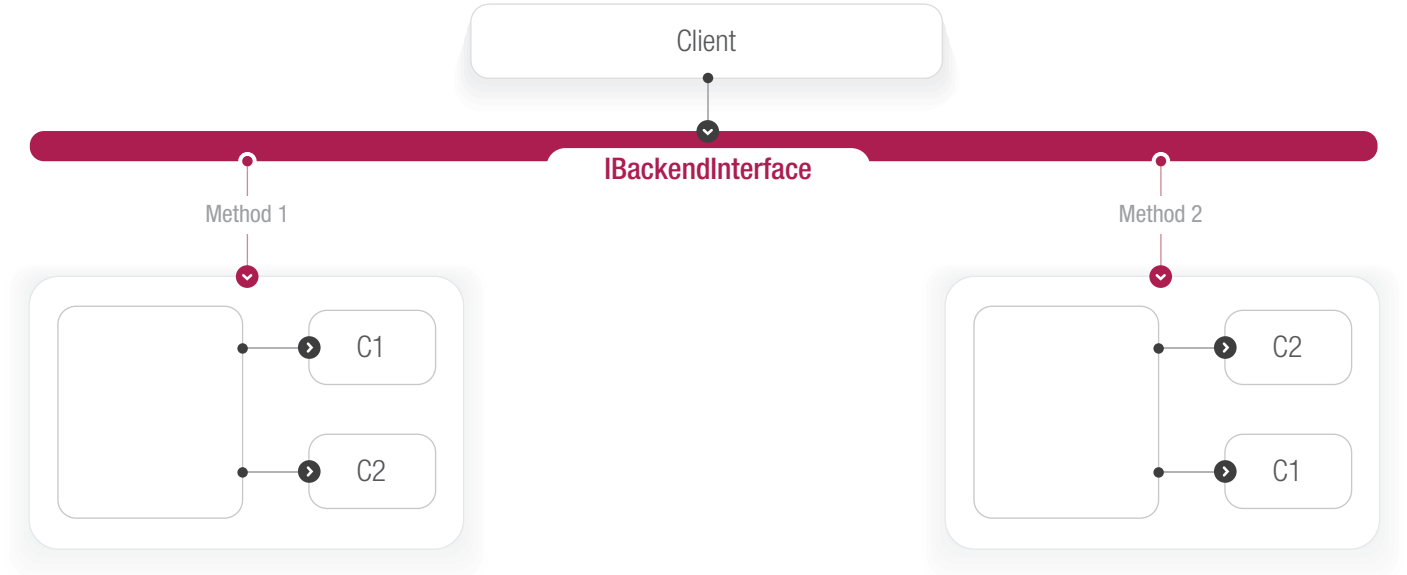


Method1 and Method2 are two workflows that share the logic hardcoded in components C1 and C2. Each method may invoke such components in any needed order, making any assumptions that is worthwhile for the final purpose. C1 and C2, though, are shared and hardly designed to be effectively shareable. Most likely, the two components have properties and logic that is the union of what is required to serve both calling methods. This is a patent example of hidden dependencies.

Dependencies are not evil—quite the reverse, as they are crucial to connect dots and build functionalities. However, when dependencies are partly hidden, not made explicit and part of an interface, they become as dangerous as rocks just below the surface. Here's a scheme to make any dependencies explicit and manageable.

Now callers know nothing about the internals of the former backend which, however, remains deployed and mostly unaltered. (It can even go to a Docker container if specific framework versions and libraries are required.) All that clients needs to know is the structure of the facade interface. The code that serves Method1 and Method2 has been split and moved to dedicated classes, each for the specific use-case. The classes are designed as black boxes with a clear input and a clear output.

Looking at the figure you may guess that C1 and C2 have been duplicated. It's definitely an option. However, it is also the first and most immediate move



in order to obtain two neatly separated subsystems invoked by a given caller. Hidden dependencies have been now neutralized. This said, more work is still possible such as moving C1 and C2 into one or two isolated and reusable pieces. This is the beauty of designing code with method and separation of concerns.

If the legacy application is an ASP.NET application, this is the essence of the shift. Once the existing backend is broken down in isolated pieces any frontend can do the job of connecting the dots—whether ASP.NET Core, Angular, React, or whatever else. As for new or modifiable use-cases, it's simply a matter of adding an additional layer on top of the facade. The final outcome is a new modern web ASP.NET Core application that largely reuses—in full safety—the existing business logic.

It's a bit different story, though, if the legacy application is a Windows desktop application.

Teaching an Old Windows Dog New Tricks

Nowadays, desktop applications have become less prominent compared to web applications. The reasons are reflecting changes in technology, user preferences, and business needs.

The rise of smartphones and tablets has increased the demand for applications that can be accessed on mobile devices, favoring web and mobile apps over traditional desktop software. Web applications, in fact, can be accessed from any device with an internet connection and users are not tied to a specific machine and operating system.

Web applications are updated on the server, so users always have the latest version without needing to install updates manually. At the same time, developers can fix bugs and roll out new features centrally and instantaneously. Furthermore, Web applications can be scaled more easily to accommodate growing numbers of users by adding server resources, without requiring changes on the client side. Security measures can be managed centrally on the server, reducing the risk of vulnerabilities compared to managing security on individual client machines. Last but not least, users expect modern, responsive, and interactive user interfaces, which are more easily achieved with web technologies.

Desktop applications still have their place, especially for resource-intensive tasks like video editing and gaming, but for the most part desktop applications in the enterprise world are now slated for replacement with web solutions.

X-Raying a Windows Application

Existing Windows desktop applications in .NET are typically built with either the Windows Forms or the Windows Presentation Foundation (WPF) framework. Windows Forms is an older technology introduced with the very first version of .NET in 2002, heavily relying on the Win32 API. In particular, it employs the obsolete GDI+ API for rendering UI elements and supports only standard Windows controls with limited customization and themes. Windows Forms applications are based on an event-driven programming model.

WPF was introduced in 2006 and uses the newer DirectX library for rendering thus allowing for superior

2D and 3D graphics, animations, and media. The user interface is defined using Extensible Application Markup Language (XAML), separating design from logic. Furthermore, WPF provides comprehensive styling and templating capabilities that allow for consistent and reusable designs.

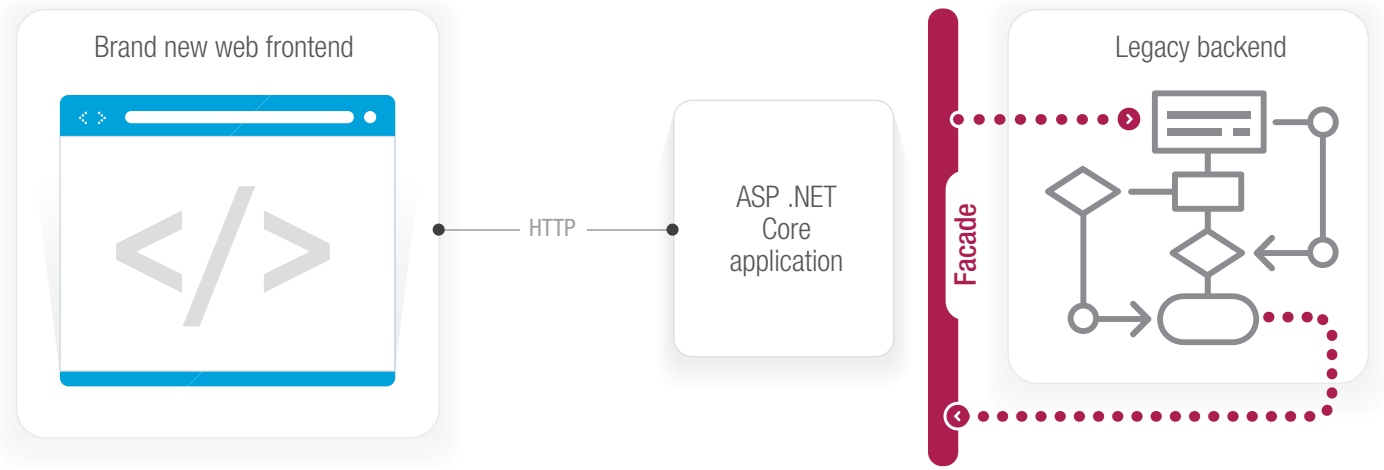
When it comes to the actual code design and implementation, differences exist: A Windows Forms application is a collection of forms. A form is a pair of files. One defines the visual structure of the form (a form is a single view); the other stores the code running behind the form. Windows Forms relies on event handlers for UI interaction. Each control fires events (e.g., Click, TextChanged) that are directly linked to methods in a code-behind class. The code-behind class is a companion file for the form and contains the logic for handling events and UI updates. UI controls are directly manipulated from the code-behind, making it quick and simpler but potentially leading to more cluttered and harder-to-maintain code.

WPF applications, instead, separate the UI, the data and business logic, and the glue that binds them. The pattern is known as Model-View-ViewModel or MVVM for short. WPF uses data binding to connect the view to the ViewModel, enabling automatic UI updates when the underlying data changes. Instead of handling events directly in the view, commands are used to bind user actions to ViewModel methods.

In both cases the user interface triggers business use-cases by invoking methods on a class. The event handlers in the code-behind classes of a Windows Forms application and the methods in the ViewModel classes of a WPF application are the root entries in the backend of the application to preserve. If the backend is an unbreakable monolith, it can be hidden in a container as is. Otherwise it can be split in pieces as discussed earlier and placed behind a facade.

Turning to a Web Programming Model

To turn a Windows legacy application into a functionally equivalent web application one more layer is necessary to handle incoming HTTP requests and mapping them to specific entry points in the legacy backend. This kind of middleware ultimately serves the purpose of covering the gap between Windows and Web contexts. The next figure provides an overall view of the middleware, where it fits and why it is needed.



The web version of a legacy Windows application has a brand-new frontend built with any framework and technology of your choice. It can be ASP.NET Core, it can be Blazor, it can be Angular or React. The frontend can only place HTTP calls to well-known HTTP endpoints. Hence, some web backend must be in place to receive requests, route them to the restructured legacy backend, and arrange HTTP responses back to the browser.

Admittedly, the task which leaves the core backend intact but demands to rewrite the presentation layer from scratch may look time-consuming but doable at first. However, rewriting the sophisticated and often clogged user interface of a years-old desktop application in HTML and JavaScript, taking into account local persistence of data, context-sensitive UI updates, modality, and extreme responsiveness of the controls makes the task quite daunting. Not a secondary point, is the reorganization of the screens through which the users would work with the application. The style of a years-old desktop application is hardly comparable to the lightweight design of typical web solutions. Each screen may need be split in two or more small screens adding issues such as routing, session data, and determining the most appropriate navigation path for the users.

The Wisej.NET Rewrite Approach

Wisej.NET is a framework that exists since 2015. At first, it may look like yet another library of controls and components for allegedly rapid ASP.NET-based web development. This is definitely true, but not entirely.

Wisej.NET is a full platform and ecosystem tightly integrated with Visual Studio aimed at building and debugging ASP.NET Core applications. To qualify an otherwise conventional ASP.NET Core application as a Wisej.NET application, you need to attach a custom piece of middleware in the application's startup. Wisej.NET supports various types of web templates that ultimately result in prepackaged projects for a number of scenarios: plain web application but also web desktop application, web page application, and user control library.

In which way is Wisej.NET different and relevant here in teaching an old Windows dog some new brilliant tricks?
The programming model provided by Wisej.NET mirrors the desktop Windows application development model. In light of this, writing a native Wisej.NET application comes much smoother if members of the team have prior experience in Windows Forms or WPF programming and are just looking to shift towards web application development. Wisej.NET simplifies numerous intricacies linked to conventional web development tools like HTML, CSS, and JavaScript. By design, the result of this effort of streamlining development is a programming model that closely resembles Windows Forms.

Even though Wisej.NET is a component-centric framework to craft modern and up-to-date web applications, it deliberately abstracts the reality of HTML, CSS, and JavaScript coding to a higher-level programming model, more centered on a plain business-oriented action/reaction scheme rather than on deep knowledge of the implementation details of today's mainstream front-end technologies.

A Wisej.NET application is based on a form container that is a collection of visual elements each of which

holds a physical position within the container and exposes a number of visual and nonvisual properties. The Visual Studio dedicated environment supplies a toolbox of components to create any necessary markup in a visual way. Each client/server interaction takes the form of an event triggered on a visual component that some handler will process.

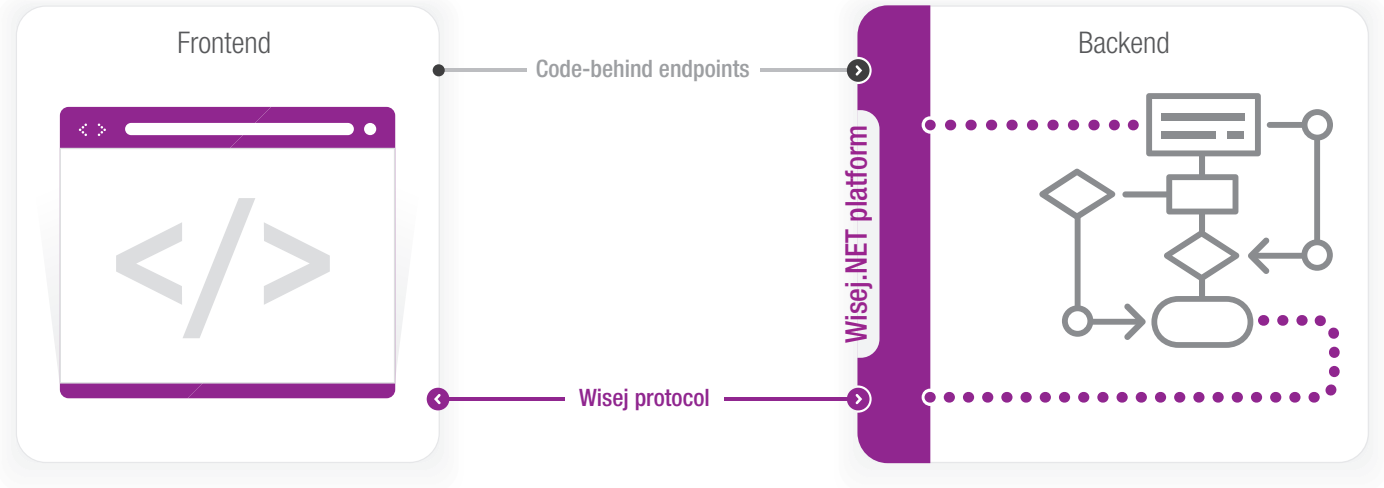
All handlers are programmed in a companion class file that goes hand-in-hand with the form container. This code-behind class has the same relevance of a controller class in a plain ASP.NET Core MVC application. Put another way, the set of all code-behind classes of a Wisej.NET application form the presentation layer of the new web application. From there, you can simply connect the public endpoints of your existing Windows Forms application backend or create a bunch of new layers (e.g., application, domain, infrastructure as in domain-driven design).

In a nutshell, once you have built a facade around the legacy backend by creating an ASP.NET Core application using Wisej.NET you're pretty much done. Have a look at the figure below.

Compared to the previous figure, now the frontend is still a web application but written with Wisej.NET. It is based on a programming model abstracted away from the raw metal of HTML and browsers' DOM. Writing a web frontend with Wisej.NET, starting from a Windows Forms or WPF frontend, takes a fraction of the time it would require with more canonical web tools.
Wisej.NET injects some crucial middleware in the pipeline that captures events fired by UI controls and—much like methods of a classic ASP.NET controller—wires the call to an endpoint exposed by the restructured legacy backend. Furthermore, the proprietary Wisej.NET web sockets-based protocol ensures instantaneous updates to the user interface after any server interactions in real-time. Wisej.NET provides responsive features, theming, and cross-platform capabilities with Wisej.NET Hybrid.

Picking the Appropriate Web Technology

So, you have a legacy application and need to make it modern and functionally up-to-date. Coming from the Microsoft stack, which technology options do



you have, concretely? For the largest part, the choice reduces to how you are going to build the frontend and which tools you use for it. Given the extreme importance of preserving line-of-business running logic, the backend consists inevitably of a sandbox to neutralize the big monolith, or in more fortunate cases, a few, reasonably large, pieces of it. Some additional layer on top of this sandbox would make it possible to receive commands from a web server.

If the backend of a legacy application should remain intact, as much as possible, the frontend—an equally critical part of the application—must change as much as possible to contribute the feeling of modernity to users and make any interaction leverage modern trends and patterns. Overall, there are not that many (raw) options.

- Large JavaScript-based frameworks
- Plain ASP.NET Core
- ASP.NET Blazor

Any of these is a considerable option if the plan is to build everything from scratch limiting the vendor lock-in to the respective developers' community and spending zero in licenses of any sort.

Large JavaScript Frameworks

The former option entails using Angular (or maybe React/Vue). Although with a few unique characteristics, all these frameworks are for building single-page applications (SPA) completely detached from the backend. Frontend and backend, in fact, are distinct applications communicating via HTTP REST or GraphQL. Blissfully skipping over arguable points such as the steepness of the learning curve and the rate of adoption in the industry, the most painful point of such frameworks is SEO.

SPA frameworks like Angular and React provide a structured approach to development, which may be a phenomenal argument to put on the table in case of large teams and large applications. But there are some substantial flaws, too. Such frameworks grew to be an ad-hoc browser within the browser turning the real browser into a mere container of runnable code.

Unfortunately, a few types of browsers never learned to work well with SPA applications and not even seem committed to it in the short term. They are primarily crawlers and screen readers. Search engine crawlers, like Googlebot, initially see an empty or minimal HTML structure. Also due to client-side routing, it's hard to understand structure and content of the site and index it properly. SEO means business and business means money. However, for most line-of-business applications it's preferable to not be indexed by Google or other search engines.

Furthermore, It's quite common for such SPA projects to accumulate directories filled with gigabytes of JavaScript packages and configuration files. This is the direct consequence of having a composed environment in which every item comes autonomously and fully configured with its own set of dependencies. As a result, not just the bundle to download is sizeable but so is the abstraction layer resulting in longer loading times, especially for initial page loads.

Plain ASP.NET Core

ASP.NET is more than ever an excellent option for backend development, but frontend developers value a physical separation between the frontend and backend concerns in web development. Hence, the trend in web development has shifted towards building

SPAs, and SPAs often rely on frontend frameworks. In some cases, using a specialized frontend framework or library can lead to better performance because these tools are designed to handle certain frontend tasks efficiently. Some examples are hot module replacement, component-based architecture, and a reactive programming model. ASP.NET, being a full-stack framework doesn't provide out-of-the-box facilities for complex client tasks and leaves everything to the capable hands of full-stack developers. This said, though, it should be noted that features like those examples mentioned above are primarily developer-friendly and not necessarily user-friendly. ASP.NET is often perceived as heritage of the Microsoft-centric past of the 1990s that the software industry at some point decided to forget. The reality is a bit different: Redesigned for .NET Core, ASP.NET is a great framework that adapts the client/server mindset to the web of the new millennium. Its older brother, though, the venerable ASP.NET Web Forms, gained a wide adoption, shaped the e-commerce business, and still has a huge installed base. 20 years later several component vendors are still making most of their revenues out of legacy ASP.NET packages. All these line-of-business applications are in need of a facelift nowadays.

ASP.NET has one crucial advantage over SPA and large JavaScript frameworks: In addition to being immensely more compact and trivial to set up, it is founded on server-side rendering which is the new frontier of SPA frameworks. To address SEO and performance issues, client-side SPA frameworks introduced yet another layer of complexity—server-side rendering (SSR). In the context of SPA applications, SSR refers to dynamic generation of the HTML page that is sent from web server to browser. Within SPA applications, this is a necessary trick, but it is also a kind of weird request for frameworks that make a point of running entirely on the client side. So Angular Universal is an extension to provide an API for developers to control the generation of HTML for tasks like data fetching and authentication. In the React camp, to set up SSR you typically need to configure the Node.js server to handle SSR requests, set up routing, handle data fetching, and use libraries like ReactDOMServer and StaticRouter. Or add yet another large framework to the heap and go for the 3rd party next.js library.

With ASP.NET, server-side rendering is plain and simple, a natural approach. To develop a large application with ASP.NET, you need a lot of self- and team-discipline instead of blindly adhering to the programming discipline enforced by the framework. The table below compares the approach of SPA frameworks and plain ASP.NET Core when it comes to techniques to updating the user interface dynamically, style and discipline of programming and supporting libraries.

ASP.NET Core	Features	SPA frameworks
Handmade CSS and JavaScript	Dynamic UI Updates	Packaged in UI components
Code-by-example	Programming style	Framework-specific programming model
Self- and team-enforced discipline	Programming discipline	Framework discipline
In-house or open-source projects	Supporting libraries	Bundled with the framework

To cut a long story short, choosing ASP.NET Core you choose to hit the bare metal of the web. You choose and craft your own tools and do things exactly as you like and want them to be. It's pure programming freedom and freedom comes at the cost of some discomfort and extra effort. But if you know how to deal with it, the outcome is paradisiac. Conversely, a SPA framework requires additional forehand learning. Not just you need to know enough about the bare web metal but you also need to immerse yourself (and the team) in the waters of the framework and learn to play (correctly) by their rules, no matter how plain or intricated they can be.

ASP.NET Blazor

ASP.NET Core, SPA frameworks, and Blazor do the same job: building web applications. Each framework, though, pushes a different and unique perspective and brings its own tools. ASP.NET is server-side and uses C# and a custom markup language. Angular/React do primarily client-side rendering and are use dialects of JavaScript for coding. ASP.NET Blazor enables the use of .NET across both client and server, promoting code reuse, and especially offering a familiar development environment for .NET developers.

With Blazor developers build web applications using C# and .NET instead of JavaScript. Blazor leverages WebAssembly (WASM) to run .NET code directly in the browser, providing a full-stack development experience with shared code between client and server. There are two main hosting models for Blazor applications: Blazor Server and Blazor WebAssembly.

Blazor Server runs on the server, with UI updates communicated to the browser over a SignalR connection. This model benefits from fast load times and reduced client-side resource usage but requires a persistent connection to the server. Blazor WebAssembly, on the other hand, runs entirely in the browser on WebAssembly, allowing for offline capabilities and eliminating the need for continuous server connectivity. This model may have slower initial load times due to the need to download the .NET runtime and application binaries.

Blazor Hybrid is a recent addition, allowing Blazor components to be integrated into native desktop applications using frameworks such as .NET MAUI, WPF, and WinForms. This hybrid model enables developers to modernize existing applications and create new ones with a unified codebase.
At the very end of the day, Blazor is a direct Angular competitor featuring a complete different set of technical aspects. From a development perspective, it sits atop plain ASP.NET Core making it faster building sophisticated, reactive, and context-sensitive user interfaces, as required by modern web applications. At the same time, when it comes to making an effective choice for a project, the relative youth of Blazor is in contrast to the success stories of Angular projects over the last decade.

Wisej.NET

As mentioned, Wisej.NET is a web development framework fully integrated with Visual Studio, with a mission captured by three key objectives. Listed in order of relevance, they are:

- Build line-of-business web applications
- Migrate existing .NET applications to the web and cloud

- Modernize existing applications to be web and cloud-native

The declared objectives of Wisej.NET embody the essential steps of modernization we discussed earlier: simpler lift-and-shift, lift-tinker-and-shift, and comprehensive (savvy) rewrites. You can assemble Wisej.NET applications using various pre-built controls and components within the familiar Visual Studio environment. This accelerates development and ensures design consistency. The resulting application typically follows a single-page application architecture, where only parts of the page are updated dynamically, leading to a smoother user experience. Notably, in-depth knowledge of front-end technologies such as HTML, CSS, and JavaScript is not strictly required, which is advantageous for teams more comfortable with back-end technologies and looking to avoid deep front-end involvement.

Additionally, Wisej.NET supports real-time updates, enabling the creation of interactive applications that can refresh in real-time without full page reloads. This behavior closely mirrors that of an ASP.NET Blazor application. In a Blazor application, after an initial handshake between the browser and the backend, there's a continuous exchange of WebSocket packets carrying request details and DOM updates. This approach enhances the user experience by providing seamless and responsive interactions.

Keep in mind that two decades ago, at the dawn of .NET and the rise of business web and e-commerce, the standout feature of ASP.NET was its ability to simplify the transition for client/server developers to web development by concealing front-end technologies and reducing the need for their expertise. Today, twenty-five years later, Wisej.NET revitalizes this approach, adapting it to the modern, cloud-intensive web environment.

In contrast to the early 2000s, rapid application development now requires more than just a visual designer and a set of standard controls. Today, a comprehensive and varied collection of controls that incorporate the latest visual enhancements is essential. Additionally, there must be built-in support for integrating external widgets, handling errors, ensuring accessibility, adding animations, and implementing text-to-speech and speech recognition features. A theme builder is also necessary to swiftly and flexibly prototype an impressive user interface, along with robust solutions for automated web UI testing. Finally, the development effort must translate into various outputs, with a strong emphasis on mobile platforms such as iOS and Android.

A modern web application can be built from scratch using a variety of web development technologies on any platform. However, there is a significant difference between starting a new greenfield project and modernizing an existing line-of-business application that is critical to the organization. For a large modernization project, failure is not an option; if the project does not reach a successful conclusion, the repercussions on management and the organization can be severe. Additionally, failure is not option and the time to complete the project cannot be indefinitely long.

Therefore, the final decision on which technologies to adopt and the modernization strategy to employ

should be based on a thorough evaluation aimed at minimizing both the development effort and the learning curve for all involved teams. The figure below illustrates the positioning of ASP.NET Core, JavaScript SPA frameworks, ASP.NET Blazor, and Wisej.NET on an XY scale, with building effort and learning effort as the coordinates.

Of the considered options, Wisej.NET is by far the one that requires the least effort to learn and build trendy applications. Compared to the others, though, it may have a license cost and being a commercial product, it may generate a form of vendor lock-in.

The Thorny Point of Vendor Lock-in

Vendor lock-in occurs when a customer becomes so reliant on a vendor for products and services that switching to another vendor involves significant costs, whether actual or perceived. Vendor lock-in is a major concern for managers dealing with large software projects, as they worry about becoming trapped by a vendor. To avoid this, they often prefer in-house or open-source solutions to maintain greater flexibility.

Vendor lock-in is certainly a concern, but it may also represent a strategic choice with its own advantages. The main advantage is the time saved in development. Using pre-built systems can shorten your project timeline by months or even years, which is essential for a quick market entry. Vendor platforms may also provide specialized technologies and features (e.g., AI extensions, automated testing, visual designers) that would be expensive and complex to develop on your own. One more point is that vendor solutions may manage essential infrastructure, scaling, reliability, and security, letting your team focus primarily on innovation and user experience.

However, depending too much on a single vendor carries risks, such as cost increases, loss of features or support, and company shutdowns. These risks are more acceptable for a startup than for an established organization embarking in a modernization process. Hence, to mitigate risks, companies should prefer vendors that support open standards, ensure interoperability with other frameworks, and allow integration with external components.

Let's now examine the potential impact of vendor lock-in when adopting the web technology listed above.

ASP.NET Core. First and foremost, ASP.NET Core is a free, cross-platform, and open-source framework. It also sports a highly modular architecture making it technically possible to replace native pieces with custom rewrites. While Microsoft plays a central role in developing and steering the direction of ASP.NET Core, it does so in conjunction with the .NET Foundation. The .NET Foundation supports and helps govern the project by facilitating community involvement and maintaining open-source principles.

Angular/React. Angular is an open-source framework developed and maintained by Google and the Angular community, licensed under the MIT License. Likewise, React is an open-source library overseen by Meta and the React community, also under the MIT License. This open-source status enables developers to freely access, modify, and distribute the source code. Both frameworks benefit from community contributions and improvements, offering transparency and flexibility in their usage.

ASP.NET Blazor. Blazor is an open-source framework developed by Microsoft licensed under the Apache License 2.0, which allows developers to access, modify, and distribute the source code freely.

Honestly, none of the above frameworks presents a significant lock-in risk with regard to the major tech giants supporting them. What about Wisej.NET, instead?

Wisej.NET. Wisej.NET is a commercial product and provides its services through a proprietary framework. Wisej.NET, though, is only a framework employed in the context of a large application being modernized. It's up to the team to consider strategies to preserve flexibility:

- Designing the application in a modular way
- Evaluating how well Wisej.NET integrates with other technologies or platforms
- Intelligently balance the level of dependency on Wisej.NET-specific features or APIs

The more you get connected, the more benefit you get in terms of speed and effectiveness of development and the more you're making your system specifically designed for a framework that would make challenging to migrate away in the future and refactor to adapt to a different technology stack.

Even though Angular and React are not commercial products, using them in a web application still presents significant vendor lock-in concerns due to the large amount of framework-specific code they generate. Migrating an application built with Angular

or React to a different framework requires extensive refactoring, and with a large codebase, it could even necessitate a major rewrite. Despite efforts to design the application in a modular way, the pervasive nature of these frameworks creates a strong ecosystem dependence, complicating migration. Thus, open-source frameworks like Angular and React can pose similar challenges to proprietary commercial platforms. Quite interestingly, in some cases, a vendor company might be more inclined to offer effective support to facilitate migration than a developers' community.

Among all the frameworks we evaluated for modernizing a web application, ASP.NET Core presents the least risk of vendor lock-in. This is because ASP.NET Core is the most generic of them all and has the least amount of abstraction built over raw HTML and HTTP. As a result, transitioning from ASP.NET Core to highly structured frameworks like Angular or React is generally simpler and more feasible compared to migrating in the opposite direction.

Another important factor to consider is the learning curve and the skill set required. Angular and React have a steeper learning curve compared to ASP.NET and even Wisej.NET. This steep learning curve isn't necessarily a problem in itself—it's a one-time cost, or it's not an issue at all if the team is already proficient. However, it becomes a significant concern from a migration perspective.

A steeper learning curve can indicate potential vendor lock-in. Developers who specialize in Angular find it difficult to switch to React and vice versa. Due to the high level of specialization and extensive features, Angular and React resemble entire platforms rather than mere frameworks. This rigidity and need for specialized expertise can limit flexibility and increase dependence on a specific technology stack. In contrast, developers experienced in ASP.NET Core and Blazor can transition to Wisej.NET with minimal effort and switch back just as easily. Additionally, these developers can move to Angular, accepting the associated learning curve, enhancing their versatility.

In essence, while Angular and React offer powerful capabilities, their steep learning curves can signal increased dependency on specific skills, making adaptation or migration to different technologies more challenging and costly. The risk of vendor lock-in is therefore higher when choosing an open-source framework with these characteristics, compared to opting for a commercial product that features rich visual tooling but a programming model closely aligned with classic ASP.NET and popular Windows models.

CHAPTER 4

Applications for the Next Decade

> *Beware of bugs in the above code; I have only proved it correct, not tried it.*
>
> **Donald Knuth**
>
> The Art of Computer Programming, 1968

When envisioning the software applications of the next decade, we routinely think of increasingly intelligent, intuitive, and seamlessly integrated tools that leverage cutting-edge technology to provide immersive user experiences. The level of automation and intelligence we achieve will clearly represent a significant departure from the present, driven by a breakthrough event or technology.

As humans, we have always dreamed of creating artificial beings capable of reasoning and thinking more profoundly than ourselves. Human literature is rich with such fantastical characters, first appearing in ancient Greek tragedies where a device (later termed deus-ex-machina by the Romans) was sometimes used to represent the intervention of a god—an entity able to resolve conflicts too complex for humans to settle. This desire for a machine that can solve our problems at zero cost is deeply rooted in human nature.

In contemporary fiction and literature, we've encountered various supercomputers designed to process any kind of data and generate results understandable to humans. A notably popular example is HAL 9000, the computer that controls the spaceship Discovery in the film "2001: A Space Odyssey" (1968). In such books and movies, human characters simply "load data into the machine," whether it's paper documents, digital files, or media content. The machine then autonomously processes the information, learns from it, and communicates back with humans using natural language. While these supercomputers were once purely the stuff of science fiction, the advent of large language models (LLMs) has made it possible to create real-world applications that not only facilitate smooth and natural human-computer interaction but also bring the long-held dream of "loading data into the machine" to a remarkable reality.

This is the era of generative AI, and this major breakthrough will undoubtedly shape the nature of future applications.

Not Any New Technology Is a Breakthrough

Dictionaries typically define "breakthrough" as a major discovery or event that advances a situation or provides a solution to a problem. In a more specific technological context, a suitable definition of "breakthrough" is: a new technology that enables the implementation of features and applications that weren't just possible before.

Not every new software or hardware technology is necessarily a genuine breakthrough capable of shaping the development of new applications. The media hype often leads to an exaggerated use of the term "breakthrough," with many predicting significant impacts for new software as soon as it is released.

Recognizably Real Breakthroughs

Since the 1970s, we had very few real technology shifts, namely discoveries or practical ideas that deeply influenced the successive development.

Relational databases. A type of database management system that organizes data into tables, which can be linked—or related—based on common attributes. Developed by Edgar F. Codd in 1970, the model introduced a structured approach to data organization, utilizing tables (relations), rows (records), and columns (attributes). The model relies on the

structured query language (SQL) for querying and managing data, promoting data integrity through relationships between tables.

Personal computing. Since the 1990s, personal computing has undergone a significant shift from a focus on enterprise applications to a wide range of personal apps. This change has been largely driven by the hardware industry, which has made personal computers progressively more affordable. In the early 1990s, computers were primarily used for business, with software focused on productivity and enterprise management. Subsequently, a new wave of applications emerged for personal use, including document editors, spreadsheets, and tools for image and video processing.

Web and e-commerce. Initially, the web served as a platform for information sharing and communication. However, with the introduction of online shopping capabilities, such as Amazon in 1994 and eBay in 1995, businesses recognized the potential to reach a global audience. The widespread adoption of secure payment technologies and the growth of digital marketing further accelerated this shift. In the early 2000s, e-commerce revolutionized the business world by allowing companies to sell products and services directly to consumers online. This shift created a more competitive and convenient marketplace, leading to significant changes across various industries.

Mobile computing. Across the 2010s, mobile computing has profoundly impacted both life and business by enabling constant connectivity and accessibility. Smartphones and tablets have revolutionized personal communication, navigation, and entertainment, making information and services available anywhere and at any time. In business, mobile computing has facilitated remote work, real-time collaboration, and instant access to data, driving productivity and efficiency. The proliferation of mobile apps has also transformed industries by offering new ways to engage customers and streamline operations, reshaping how we live and work.

Cloud infrastructure and computing. Since the 2010s, cloud computing has provided unmatched on-demand access to computing resources and data storage. This capability has facilitated seamless collaboration and remote work, enabling individuals and teams to access applications and files from any location. For businesses, cloud computing lowers infrastructure costs, improves scalability, and fosters innovation by enabling the swift deployment of new services. This transformation has fundamentally altered how organizations function and how individuals engage with technology.

Conversational computing. Short for Large Language Model, LLMs—the technology behind remarkable applications like ChatGPT—introduce a third key capability to artificial intelligence (AI): generative content, alongside classification and prediction. LLMs drive a huge paradigm shift, elevating human-computer communication and unlocking a wealth of new applications that were once only envisioned in our dreams.

Powered by AI, conversational computing represents the next frontier in software applications. Today, any meaningful effort to modernize applications must include effective integration of generative and conversational capabilities.

Significant but Scope-Limited Innovations

Over time, various technologies have brought significant cultural and procedural changes to the software and IT industry, but they have not delivered widespread benefits on a large scale to the general public. Although we might not categorize them as breakthroughs, these technologies are certainly noteworthy when discussing factors that impact the modernization of legacy applications.

For instance, DevOps represents a procedural shift rather than a traditional technological breakthrough. It emphasizes collaboration between development and operations teams, streamlining workflows, automating processes, and enhancing the software development lifecycle.

Similarly, containerization is a significant but narrow-impact technological advancement. Containers encapsulate applications and their environments, ensuring consistent performance across different computing setups. Technologies like Docker and Kubernetes have become central to modern cloud-native applications and microservices, reshaping how software is developed, deployed, and managed.

Moreover, the internet of things (IoT) stands as a substantial advancement by enabling real-time data collection, monitoring, and automation. It has made transformative impacts in sectors like smart homes, healthcare, manufacturing, and transportation, signifying a major shift in how technology integrates with the physical world.

Examples of False Alarms

From a software-only perspective, it's easy to confuse advancements in frameworks with foundational technologies like the internet, cloud computing, or artificial intelligence. In the recent years, many web frameworks have been marketed as revolutionary paradigms, but in reality, they represent, for the most part, significant improvements and innovations only in frontend development. True breakthroughs have the power to drive dramatic modernization efforts, but the mere availability and evolution of frameworks like Angular, React, Svelte, or Vue shouldn't compel a company to abandon legacy applications just for the sake of adopting such new frameworks.

Therefore, if you're seeking breakthroughs to justify modernization efforts, web frontend frameworks are not the answer and sound more like a false alarm. Several other technologies have been marketed as breakthroughs—often with good reason—but have not yet significantly transformed business on a large scale. Here are a few examples:

- Blockchain
- Virtual Reality (VR)
- Augmented Reality (AR)
- Quantum computing

Although blockchain has made a significant impact in the cryptocurrency space, its applications in other areas—such as supply chain management, digital identity, and rights management systems—have faced challenges. Scalability, high costs, and integration difficulties have limited its broad-scale transformative impact. Ultimately, it has not lived up to the hype. Yet,

until about five years ago, including the word "blockchain" in an application rewrite pitch was essential to capture executives' attention and secure funding.

Despite being around for over a decade, virtual reality has not achieved the transformative scale initially anticipated. High costs, limited content, and hardware requirements have hindered widespread adoption. Similarly, augmented reality (and dedicated AR glasses) faced similar obstacles to large-scale adoption. However, AR may experience a resurgence due to integration with generative AI, which can seamlessly incorporate voice and human gestures into commands and content generation.
Quantum computing has generated considerable excitement and is viewed as a theoretical breakthrough. However, practical, large-scale business applications are still in the early stages. Current quantum computers face challenges such as high costs, technical complexity, and, for now, limited real-world use cases.

Conversational Transformation

In the next five to ten years, we are likely to witness a proliferation of AI-driven applications seamlessly integrating into various aspects of daily life. Personalized virtual assistants will become more advanced, offering tailored recommendations, managing daily schedules, and even anticipating user needs before they are expressed. In healthcare, AI will enable more accurate and timely diagnostics and personalized treatment plans, significantly enhancing the patient experience in hospitals and care facilities.

Significant advancements in augmented reality applications are also expected. These technologies will revolutionize how we interact with digital content and our physical surroundings, providing immersive experiences in entertainment, education, and training. AR will aid in tasks ranging from complex machinery repairs to interactive learning experiences with digital overlays.

The driving force behind this conversational transformation—far more immersive and specific than mere digital transformation—is generative AI. This raises an important question from a software perspective: What will be the patterns and architecture of next-decade applications driven by AI?

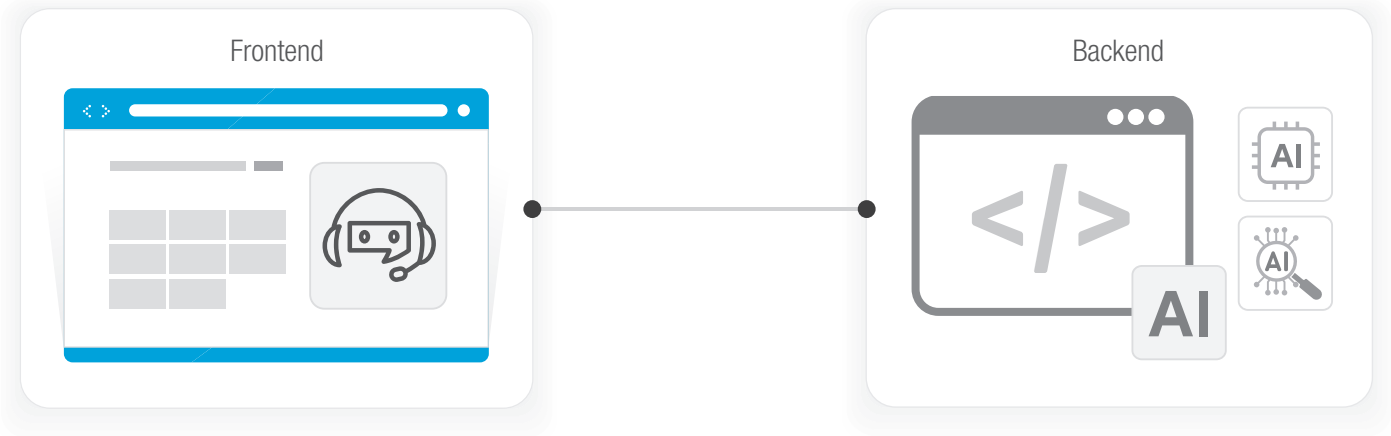
The Copilot Pattern of AI Applications

Today, AI capabilities like machine learning algorithms, natural language processing, or predictive analytics are typically built on top of existing applications and exposed as additional features. Integrating AI features into existing applications is quicker than designing and building an AI-driven application from scratch and this allows organizations to benefit from AI advancements sooner. A common practice of adding AI services to an application is the copilot pattern.

The copilot pattern refers to a development approach where AI technologies act as an assistant to human users, helping them perform tasks by providing suggestions, automating processes, or enhancing decision-making. The term was adopted just to describe the behavior that resembles that of a supportive partner, similar to how a copilot assists a pilot. A well-known example is GitHub Copilot, developed by GitHub in collaboration with OpenAI, that assists developers with code suggestions and completing code snippets. Another great example is

Microsoft Copilot available in a number of Microsoft products to assist users with tasks in applications like Word and Excel.

Microsoft Copilot is the successor to the discontinued Cortana. Copilot leverages the Microsoft Prometheus model, based on OpenAI's GPT-4, and has been further refined through supervised and reinforcement learning. Its conversational style is similar to ChatGPT, and it can cite sources, compose poems, generate songs, and support various languages and dialects. Throughout 2023, Microsoft consolidated the Copilot branding across its chatbot services, reinforcing the copilot concept.
The figure below illustrates two potential schemas for a Copilot-based application. In one scenario, the Copilot service is fully embedded within the application's user interface as an integrated component. In the other scenario, the Copilot operates as an external service, interacting with the main application through an API.

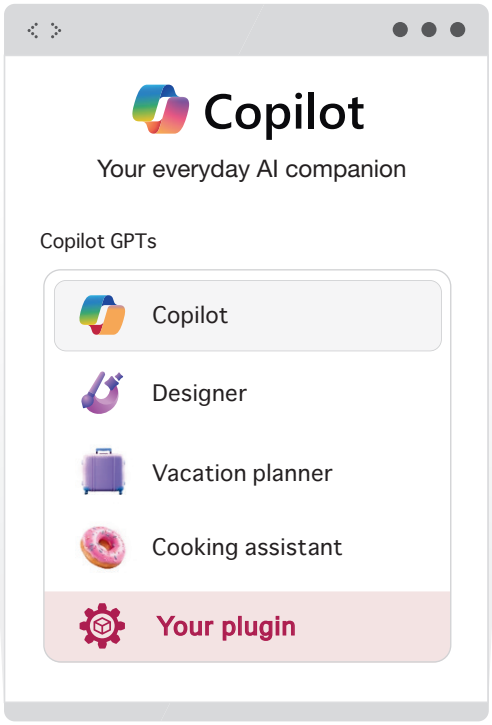


Copilots give their best when combined with Retrieval Augmented Generation (RAG), an information retrieval technique that enhances interactions with large language models by incorporating a specific set of documents in response to queries. Instead of relying solely on its extensive, static training data, the model leverages the specified documents and custom data to provide more relevant and contextual answers.

A Brand-New Flavor of Applications

What does the future hold for AI in applications? Will AI remain as a smart add-on for traditional web or desktop applications, or are we on the verge of seeing entirely new types of applications emerge?

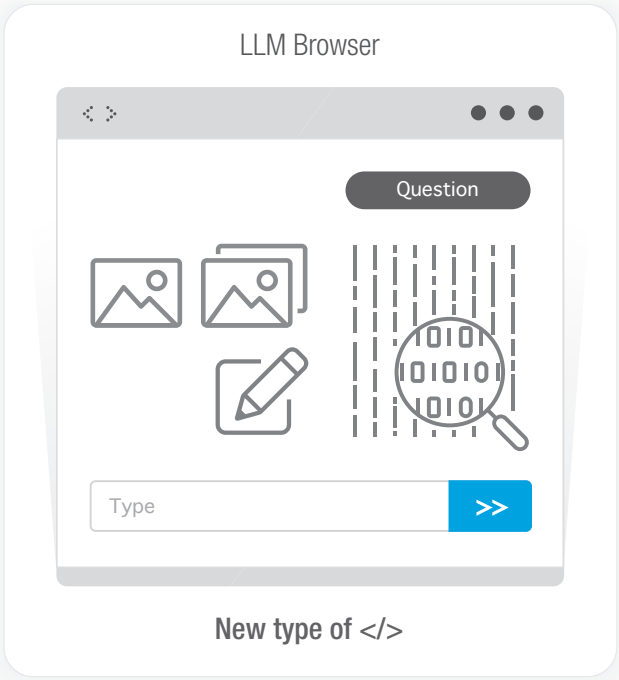
A potential preview of an alternative future can be seen on the copilot.microsoft.com website.
Once the user logs in, the website offers a list of available plugins that can be composed together to form a user-specific dashboard. Currently available plugins include frames of popular travel, food, and shopping websites. The rest of the site is a search application that uses any of the linked plugins as a data provider. Overall, the model is still quite rudimentary, but you can get a sense of something emerging. The figure below presents a glimpse of a possible future new generation of applications.



Currently, companies can develop and run dedicated plugins within the Microsoft Copilot container. If fully pursued by the industry, this approach may lead to a new Copilot-first application model in just a few years. While it's too early to place all bets on this, a familiar pattern is recognizable. Recall the early days of the web: HTML pages started as simple structured documents and gradually evolved to include small applets, a full script engine, an updatable object model, stylesheets, and eventually server-side frameworks, giving rise to a new category of applications for the web.

Similarly, we can anticipate a new type of single-container application—a sort of LLM-powered browser—that hosts sandboxed components with a common pluggable interface. These components might be orchestrated in their overall behavior by an interpretable language, as summarized in the figure.

It could be this model, a variation of it, or an entirely different approach, but expect to see radical changes in the taxonomy of software applications in the next few years. These changes will be comparable to the revolutions brought by the internet and web initially, followed by mobile and cloud computing.



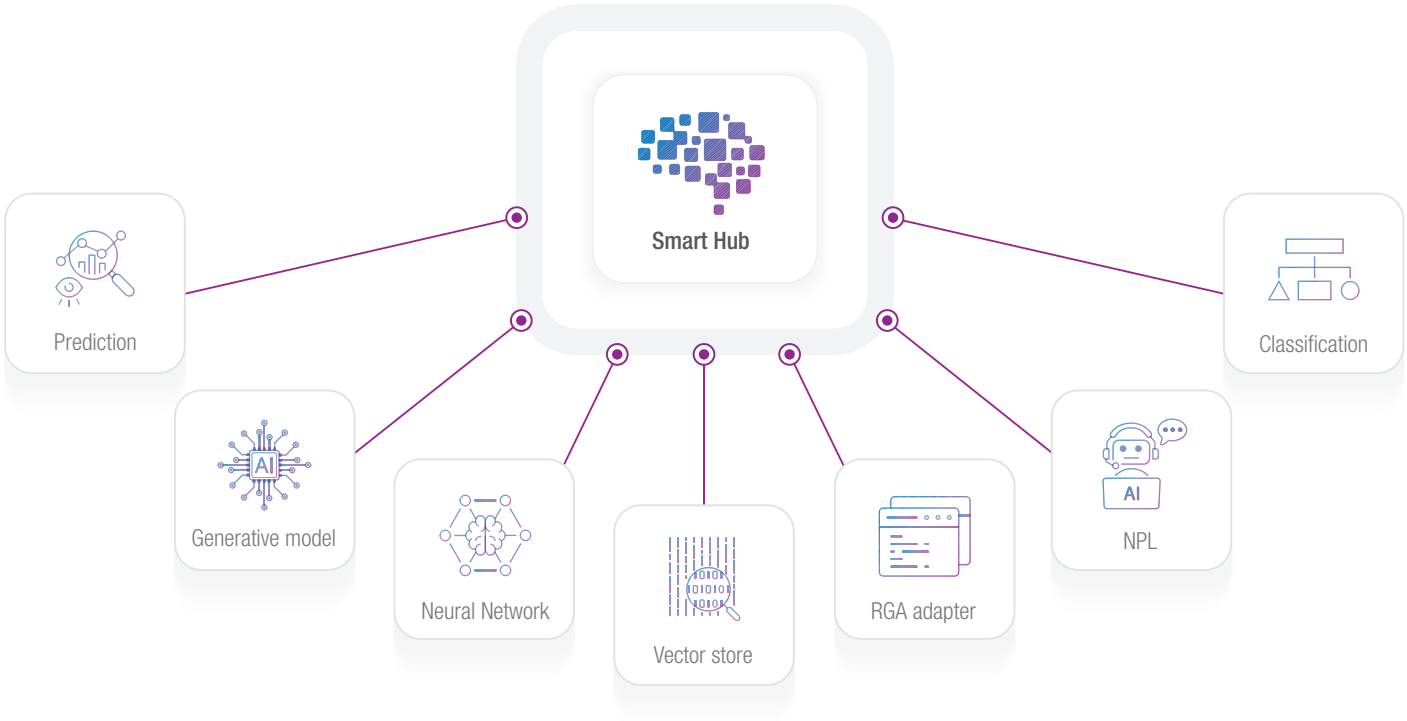
Wisej.NET AI Hubs

Whether publicly disclosed or kept private and proprietary, the Microsoft's Copilot website does have an extensibility model for partners to contribute extensions that the internal logic of the site will orchestrate. As of now, the Copilot website only allows plugins to contribute their data to a search algorithm that begins with the user's inquiry.

A similar but in many ways smarter model is available in Wisej.NET.

Wisej.AI, a companion platform to the core Wisej.NET, provides a centralized hub where developers bind together endpoints and adapters in much the same way they compose visual controls on a web or Windows form. The result is custom data flows within custom and business-specific algorithms–not just search, but also completion, natural language processing, classification, prediction, recommendation, and any sort of dedicated neural network built for a specific business domain.

The overall pattern suggests an in-framework anticipation of the aforementioned, yet potential, shift in general software applications. In this model, the Wisej.NET main application serves as the container, and the smart hub—hosted within the app—acts as the sandbox running user-defined AI workflows with custom prompts and .NET accessible data sources. Additionally, the output from any connected AI endpoint is automatically parsed and used to populate controls through specialized adapters.



Drawing the Bottom Line

While enchanting stories of groundbreaking and revolutionary frameworks dominate tech headlines and flood our online feeds, substantial value persists in a forest of legacy applications built upon steadfast frameworks that have stood the test of time. These line-of-business applications, as unremarkable as they may seem, sustain daily operations and embody years of accumulated wisdom and expertise. Many of these applications were constructed based on the .NET Framework and continue to run reliably on both aged desktop PCs and the latest laptops.

Legacy systems are essential to most enterprises, providing business continuity and vital revenue streams. However, they also hinder organizations from adopting new digital technologies and creating innovative experiences for customers and partners. Initially, line-of-business applications were desktop-based, and many transitioned to web applications as anytime/anywhere access to business data and processes became possible. ASP.NET Web Forms was the most viable option during this shift. Over the years, these applications have undergone changes and revisions, driven more by changing business needs than technological advancements. Consequently, a considerable number of aged applications in the .NET space are bound to older versions of ASP.NET and the .NET Framework.

All these applications need to be rewritten.
Not all companies may always be aware of this necessity, but it is an unavoidable, yet painful, reality. The more time that elapses, the harder the task becomes. Updating business-critical applications, regardless of their size, is daunting but essential.

In the 2020s, for an application being web-based, at least cloud-friendly if not cloud-native, and connected to scalable services is not just a must, it is vital. Furthermore, having a DevOps arrangement for quick rollout and rollback is also necessary. And a modular and modern codebase is the only way to avoid the subtlest form of lock-in. Worse than vendor lock-in, in fact, is developer lock-in, where the uptime of a business-critical application depends on a small number of people who may leave the company at any time for various reasons, including natural retirement.

In summary, while the allure of new frameworks captures attention, the real challenge and necessity lie in modernizing the extensive landscape of legacy applications that continue to drive business value.

To successfully manage large, complex, and delicate projects, the last thing you want to do is reinvent all the necessary wheels and craft every single component from scratch. Instead, leveraging frameworks and seeking assistance from vendors is crucial. After all, software development is likely not your core business. Utilizing established frameworks and vendor support can streamline the process, reduce risks, and ensure that your projects are built on a foundation of proven, reliable technology. This approach allows you to focus on what truly matters: delivering value to

your business and customers without getting bogged down in the intricacies of software engineering. Does that mean you should close your eyes and blindly trust a proven vendor? Absolutely not.

As outlined in the section on notable failures of application rewrites, blind trust can be detrimental. In some cases, companies, recognizing that software isn't their core business, tend to place their faith entirely in vendors. While trust is fair, good, and acceptable, it is not enough on its own. Verification is crucial. Just because a vendor has a solid reputation doesn't mean they are immune to mistakes or misalignment with your specific needs. Thoroughly vetting solutions, seeking references, conducting pilot tests, and continually monitoring progress are essential steps to ensure that your project stays on track and delivers the desired outcomes.

Another critical factor is maintaining control over project development. For a large migration to succeed, neither shortcuts nor overly extended timelines will suffice. It's about having a clear understanding of what is needed and obtaining it as quickly and efficiently as possible, ensuring the internal team retains full control throughout the process. This involves setting clear objectives, establishing robust project management practices, and ensuring transparency in all stages of development. By doing so, you empower your internal team to make informed decisions, address issues promptly, and adapt to changing requirements effectively, thereby increasing the likelihood of a successful migration.

One final note on testing critical applications. Legacy software often feels intimidating to change not because of the change itself, but because we want the new version to maintain the exact current functionality. A software application becomes legacy just because it works; and keeps working even when internally it is only an outdated tangle of code. On the other hand, you can define software of high-quality when all bugs are fixed and the application is used daily without revealing new issues.

How to release a new application safely?

Typically, teams use dedicated staging environments to test applications before releasing them into production. Developers often prefer staging environments due to the control they seem to offer. However, setting up an effective staging environment involves a couple of major challenges:

Data duplication. The real production dataset must be copied over. This is not just costly and time-consuming but may also privacy and security issues. Anyway, it is doable in some way.

Effective simulation of user activity. A staging environment that doesn't closely mimic the real user activity is of little use. Realistically, though, a staging environment can only give an approximate measure of the complexities and edge cases you may face in production. What you catch in staging is as good as testers are good and committed. And, personally, I wouldn't trust it that much!

Believing that exhaustive testing can prevent all issues before release is a fallacy and the true test of resilience happens only in a live environment. As stated by Uber engineers, every deployment should

be treated as an experiment and the sooner you start it–namely skipping the staging phase entirely–the sooner you get real facts and can make steps towards real high quality software.

Therefore, testing in production, though initially unsettling, is a crucial strategy for achieving resilient and reliable services. Edge cases and thorny issues emerge only in production, and addressing them with agility and flexibility is essential. This approach requires specialized tools and policies to detect failures early and quickly roll back to a safe state while fixing any bugs.

In this context, one of the gravest errors is deploying a new version to the entire user base all at once. A more strategic approach involves rolling out updates to a small, yet meaningful segment of users first. Ideally, this group should be located within a time zone that aligns closely with where most of the development team operates. This setup facilitates real-time monitoring, accelerates problem detection, and allows for quickest possible remediation.

As unsettling as it may seem, it's about time to question the conventional dependence on staging environments and embrace a more pragmatic perspective. The application must work reliably in production, meaning that any user-triggered function should operate without errors. Ensuring that an application will perform well in production based on tests conducted in a staging environment is an illusion.

Author of this Whitepaper:
Dino Esposito

About Dino Esposito

Dino Esposito has authored over 20 books and 1,000 articles across a career spanning more than 30 years. His work has been instrumental in the professional growth of thousands of developers and software architects working with Microsoft technologies.

He began his journey in 1992 and, just a few years later, led a team of five visionaries who, in 1995, built a platform uncannily similar to what we now know as Flickr or iCloud Photos.

With two decades of experience in building software for professional tennis and live scoring, Dino has become a key player behind the scenes of the global tennis and padel ecosystem. If tournaments happen every week around the world, it's also thanks to the platforms he helped create. A former JetBrains technical evangelist and top-rated Pluralsight instructor, Dino has spoken at over 100 industry events in more than 20 countries. After a two-year detour into renewable energy, he returned to his roots in professional sports software.

Together with his son Francesco, Dino founded Youbiquitous—a fast-growing AI factory developing cutting-edge solutions in podcast generation, augmented reality, speech-to-speech architecture, and agentic AI.

Modernizing Business Applications

www.wisej.com

iceteagroup.com