merixstudio

Modernize and grow

How to prevent outdated software from slowing down your business growth

www.merixstudio.com

Table of Contents

Growth-driven modernization journey

Heraclitus, a renowned Greek philosopher, once said, "the only constant in life is change". Over 2000 years later, we can still acknowledge the truthfulness of this statement in many aspects of life, including business and software development.

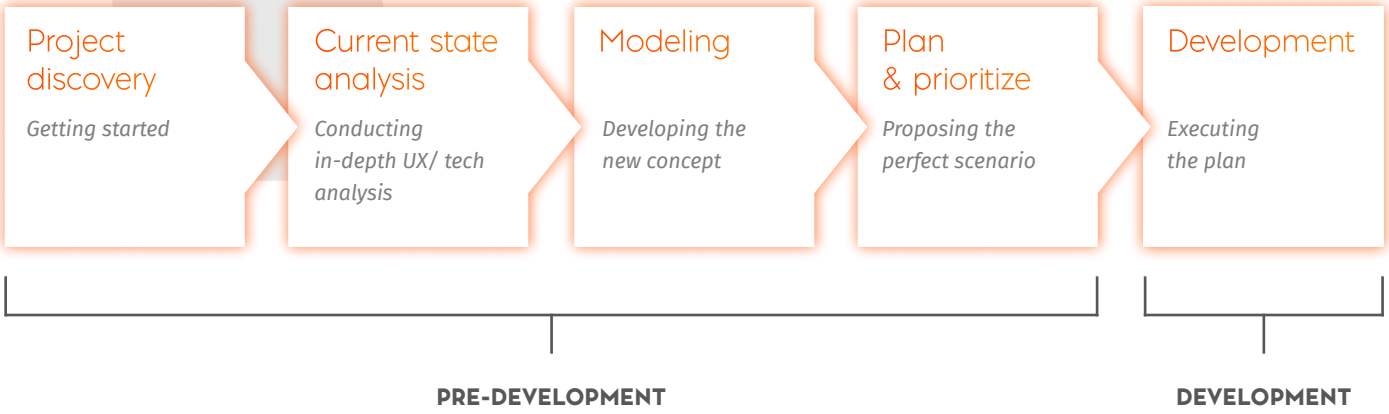
These days, you have no guarantee that the system you build today won't be at risk of becoming outdated tomorrow due to the pace of technological change, shifting skillset availability and cost, as well as changing business needs. ***The trick is to realize the need for modernization quickly enough not to let the aging application disrupt your company's growth****. This, however, is just the beginning. Many elements need to fall into place for software modernization to happen, including stakeholder buy-in, well-defined goals and needs, clear roadmap, future-proof tech choices, and skilled team – and that's not even a finished list!*

Just like your business is unique, your modernization path will be unlike any other. Speaking from our experience, we can tell you, though, that there are certain steps you can take to make it less bumpy and facilitate business growth at the same time.

Modernization is a journey. Let us be your guide, from preparation to development.

Give yourself a head start with the pre-development stage

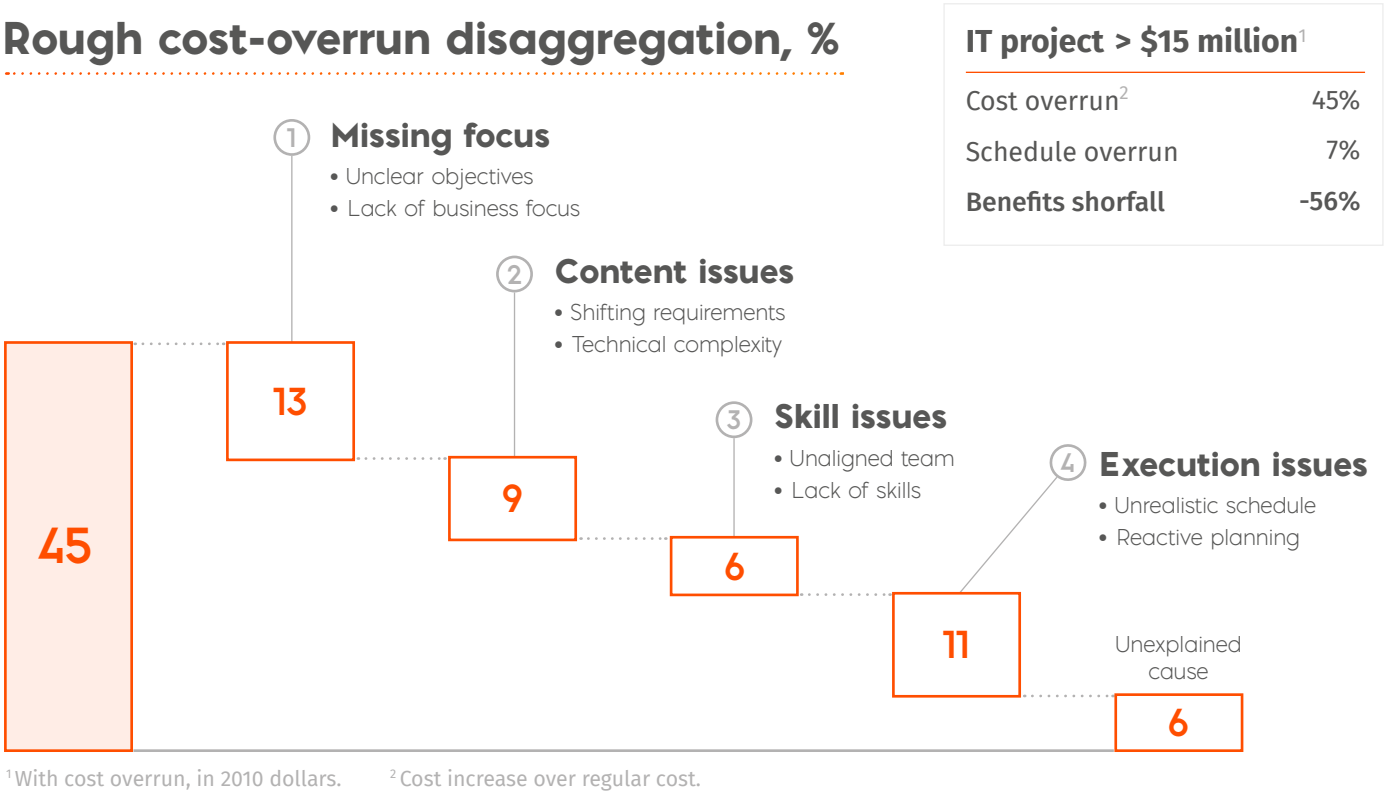
Software modernization is a journey that's not likely to go well without solid preparation. To begin with, it is necessary to look at the map to discover where you are and what processes will determine the escapade's course. Then, you need to know the gear at your disposal. Having a good overview of your future destination and the best route to take will also make the tour smoother and faster. Lastly, do not forget about the itinerary and a list of must-see attractions. In this chapter, we'll guide you through the preparation stage of your software upgrade expedition.



TAKE A LOOK AT THE MAP: PROJECT DISCOVERY

The first phase of the modernization process is project discovery, which is particularly important when you use the tech partner's support during the digital upgrade. This exploratory part allows the external software team to streamline modernization efforts and align them with your business needs. At this stage, the tech experts gain in-depth knowledge about your company, processes, and stakeholders to inform the subsequent steps of a software revamp. Its primary aim is to answer the pending question: **What kind of problem is this application supposed to solve?**

To demonstrate the significance of taking a big picture look at the company and the business problem in question, let us refer to the research by McKinsey and the University of Oxford. An in-depth analysis of more than 5400 IT projects with a budget exceeding $15 million shows a 45% budget overrun. **The primary reasons for such budgetary extravagance turned out to be the risk of having ambiguous modernization goals and insufficient concentration on the project's business value.** The combination of these two constitutes 13% of the total 45% budget overrun level.



IT executives identify 4 groups of issues that cause most project failures. Source: McKinsey-Oxford study

Business value is indeed key – not only for the discovery stage but for the whole project. Sure, the driver behind a software upgrade may be imperfect technological solutions unaligned with business tempo. Still, it's the specific business value that lies at the core of your modernization project – whether you modernize with external software expert's help or not. Let's take a look at the steps that internal or in-house software professionals take to achieve a business orientation of a software revamp, shall we?

In the course of the project exploration stage, your tech team needs to get a bird's-eye view of the system and business at hand. Before that happens, however, the following questions need to be answered:

1. **What business processes take place in your company that will influence modernization efforts?**

2. **What stakeholders' needs should be taken into account when designing a future digital solution?**

UNDERSTAND BUSINESS PROCESSES AND GOALS

To understand the processes and the **goals of software modernization**, all parties involved must know the nature of your business inside out – especially when you hire external help. Members of the modernization team need to understand what actions are part of your operations. At the beginning of the project, the Domain Experts from your side know the flow best – if you choose to collaborate with the tech partner, their help will be invaluable in creating a **Business Model Canvas capturing the essentials of your business**. The Canvas includes the activities performed to deliver the value proposition, customer segments, distribution channels, cost structure, and revenue streams. **Seeing a big picture allows the tech team to look at a modernization project from a business-oriented perspective and paves the way for profiting from the modernized solution in the future**.

One technique that supports the exploration of events within the business flow is Alberto Brandolini's **Event Storming**. Sometimes called the "smartest approach to collaboration beyond silo boundaries," it helps to identify the system's inner workings by leveraging the knowledge of Domain Experts about Domain Events.

> The upside of using the discovery-stage Event Storming is the ability to learn much about the system's processes with the benefit of stakeholders' insights.

Exploration-level Event Storming shows the process entry and exit points, focal elements (Brandolini's Pivotal Events), and hurdles to get an overview of the system.

Service blueprint is another technique used to supplement the overview of a given project's business services. According to the Nielsen Norman Group insights, it's a form of organizational flow visualization to provide an optimal UX and arrive at a **Single Source of Truth** about the service experience.

> "The blueprint for service design provides a complete picture of the organization because it shows the customer journey as they interact with the physical and digital assets of the organization."
>
> Vilas Uchil, CTO at BullsEye Telecom and Forbes Councils Member

TUNE INTO THE NEEDS OF SOFTWARE MODERNIZATION STAKEHOLDERS

When looking at the different ways in which users utilize your digital product, you should pay attention to whether the project team, be it internal or outsourced, has identified the stakeholder requirements that will shape the future solution. These considerations are important because **software experts can't achieve usability benefits while upgrading your system without having the project stakeholders in mind**.

While the specific stakeholder characteristics may vary from project to project, the below generalization captures a sample variety of their requirements you may have to address during a software upgrade:

- **Customer** – a buyer paying for the digital product, not necessarily always using it,
- **Power user** – a high functional expectations user accustomed to an intricate User Interface (UI), for whom the tech team typically creates custom-made and business-domain specific system functionalities,
- **Occasional user** – the one that utilizes a digital product or service on an infrequent basis and cares about the relative intuitiveness of the solution,
- **IT operator** – an administrator who manages the access to the system for other users and deals with account setups in this regard,
- **Feature beneficiary** – a system user that benefits from the data gathered in a given feature without actively using it through their own account.

To further illustrate the intricacies related to system users, let us take the example of a financial services company:

> Dashboarding is a data visualization tool often used in the financial context to present graphically specific business metrics such as KPIs. On the surface, you could argue that a dashboard for monitoring company performance is mostly used by Business Analysts who present their reports to the Financial Director. Yet, the Finance Director is the **feature's silent beneficiary** taking advantage of the dashboard-originating data without actually actively using this functionality. A failure to take into account this aspect would neglect the requirements of this particular user.

Aside from the business process overview and stakeholder requirements analysis, you needn't forget about a **competition analysis**, which will provide technical benchmarks for your future solution. Steve Roemerman, the Lone Star Analysis CEO and Forbes contributor, compares creating a strategy without competitive reconnaissance to preparing for a trip without using a map. Competitive insights guide you on the road to software modernization, allowing you to recognize **your digital product's strengths and weaknesses against the broader market**. The competition analysis may include, for example, a UX and features comparison.

In summary, the primary outcome of the project exploration phase is the deep knowledge of the product, process, and business needs together with the project scope. Having a firm grip on those, we can move on to analyzing the system's current state.

CHECK YOUR GEAR: ANALYZE THE SYSTEM'S CURRENT STATE

Defining the system's current state is an indispensable part of software modernization. You won't be able to effectively proceed with an upgrade unless you know what had originally shaped your software. How to approach the system's analysis to effectively progress towards a well-modeled future solution and not get stuck amidst the challenges posed by mature applications? In the following section, we examine the key factors to be accounted for when discovering the status of your software. **They are centered around a UX and technological analysis of the system**.

BENEFITS OF USER-CENTRICITY FOR ACHIEVING BUSINESS GOALS

Our experience in creating digital products shows that the user should be one of the central points of reference when gaining insights into the system's current state. So, how exactly is an external perspective beneficial for your business?

Software modernization, like any business-affecting endeavor, should serve as a means to achieve business goals. Where better to look for the realizations of these aims, if not at the user's path? The product is only part of the equation. What you also need to examine is the actor's trail across the whole ecosystem. At this point, the **user-centered design (UCD)** comes in handy. The UCD approach to system analysis holds substantial benefits. First of all, it boosts your prospect of sales gains from satisfied customers. Secondly, the more user-centric your product is, the lower the risk of its redesign and further costs, and a higher competitive edge for the future. In his famous quote, "Beware the inside view", Daniel Kahneman, the 2002 Nobel Memorial Prize in Economic Sciences laureate, warns of the risks that the inward orientation can pose to a project. Software modernization presents a similar analogy in that an **external view helps companies better position themselves against their business challenges**.

The user-oriented investigation into the current version of your system typically involves a product performance and usability analysis based on **UX research and interviews** and a **UX audit**. When looking for quality outcomes of a UX audit, you may want to pay attention to whether the team performing it uses the specific insights from your business as an input for it.

BUSINESS INSIGHTS USEFUL FOR A UX AUDIT

- ***Using your customer service or helpdesk to capture the systems' functional deficiencies***

  Your customer service system is a repository of insights on what does and doesn't work when it comes to serving your clients. It would be best if your software team used this knowledge to gain quality input for a UX audit.

- ***Leveraging the power of data-driven website traffic insights using Google Analytics***

  Google Analytics provides you and your prospective software partner with quantitative website data, including drop-offs and other traffic stats. By gathering these, a UX team can determine the specific problematic hot spots in your UX.

- ***Performing usability tests with user participation to examine internal software***

  Usability assessment makes software professionals aware of the profile and behavior of your users, their opinion of your product, and your application's user experience.

- ***Diving into user behavior analytics with remote UX research tools like Hotjar or Clarity***

  These heatmap tools allow the software team to follow your user's path during a given session, which helps identify areas of activity and non-activity.

It is equally important that your software specialists provide you with **specific areas for improvement** after such an in-depth assessment is complete. Moreover, metrics such as user engagement and conversion rate should be used to measure the success of these enhancements. As a result, what you get as an IT decision-maker is a set of actionable insights that should enable better software performance through usability fixes and application redesign (if necessary). What's more, by analyzing the results of user interviews, you may be able to discover the missed opportunities that, once addressed, will elevate the UX level. All of these directly impact the customer service experience of your application and your revenue. To illustrate the **practical value of a thorough UX audit**, let us examine a case study where the software team had used it.

Leo Trippi is a Swiss luxury travel agency providing ski holidays and personalized escape retreats. Their original software setup consisted of codependent WordPress-based applications like Pricing and Availability Calendar Application, a Sales Dashboard, or the main company website. These solutions had yet to match a business tempo set by the company. Therefore, Merixstudio implemented a system upgrade.

The software used was aimed at providing a more systematic architectural structure and website redesign. Python and Wagtail ensured the tech stack's scalability for future business growth and provided one source of truth about Leo Trippi's luxury holidays offer. REST APIs were used for data distribution, enabling data coherency and unimpeded communication between the new ReactJS-based applications. **The usability audit served as input for the website redesign, where the tech team made improvements in terms of information architecture, navigation, and mobile UX**. Lastly, the applied AWS infrastructure brought about reliability, security, and performance gains. The final system that took eight months to develop supports the client's business offer, including approximately 750 luxury properties.

The UI and UX audits and subsequent improvements applied in the above case translated to the quality of the design experience and informed the system-internal information architecture.

TAKE A LOOK INSIDE THE SYSTEM WITH TECHNOLOGICAL ASSESSMENT

Apart from the usability analysis, the other step to be taken is the technological assessment. It aims to examine your system from the technical side, including architecture, solution-critical areas, and the implementation technology. While the scope of this analysis is relative to an individual case (both your representatives and external developers usually attend such a discussion if you hire external help), it will typically include:

- **architecture review,**
- **development and QA processes review,**
- **integrations review,**
- **deployment review,**
- **technological audit (with its constitutive parts).**

One of the benefits of performing the system's tech evaluation is that it allows you to categorize system parts to fit specific modeling techniques that ultimately save you time and money. Let's take a look at the different system components' categories that may occur then.

To classify the system into subdomains, software professionals may refer to **Domain-Driven Design (DDD) used customarily as a strategic methodology** during software modernization. We may categorize the system components as follows:

- **core system subdomain,**
- **generic system subdomains,**
- **optional system subdomains.**

Particular emphasis in this approach is placed on the fact that generic system subdomains can be modeled through ready-made solutions based on the available software modeling archetypes. Such subdomains do not require tailoring to specific business requirements, contrary to the core system components that should be developed in a customized manner. Finally, optional system subdomains are secondary-importance components. The DDD approach to the system's technological analysis holds crucial advantages: it's a shortcut that saves your time, money and provides focus on the core system domains that will later fundamentally support your business.

> We recommend using the DDD strategic methodology to identify the core domain as well as generic and supporting system subdomains.

LEARN FROM YOUR EXPERIENCE: SYSTEM GENESIS

Apart from the domain classification, during the as-is assessment, you should learn about the **genesis of the technological status quo of your system**. As is often the case, the circumstances that once shaped a given organization's technology choices may have changed but will nevertheless affect the newly-developed solution. What are some of the scenarios that may occur here?

THE GENESIS OF THE TECHNOLOGICAL STATUS QUO: POSSIBLE SCENARIOS

The system technology applied in the past corresponds to the then-relevant skill set of the internal software team

The past decades tipped the scales of technologically-oriented software modernization efforts, as it had been the case approximately ten years back, towards a more business demands-driven approach. This new basal attitude means that technology is no longer supposed to fit your team but rather release the trigger holding you back from increased business growth.

- ***Mature software may have been shaped by limited budgetary perspectives of the previous financial years***

  Your current business may present enough opportunities for you to embark on the road to modernization, including altered financial perspectives. The new budget should be aligned with your altered technological demands as well.

- ***Blocked business and the need for an immediate solution may have influenced your past software choices***

  While right now, you probably can no longer afford a quick fix, knowing the reasons for the as-is state, software professionals can adjust their current actions to a more long-term, scalability-driven strategy.

That being said, none of the parties participating in the modernization efforts should fall into the trap of the commonly made mistakes at this stage which can be:

Hype-driven development

As our software development expertise shows, it pays off to avoid falling victim to software development driven by the most popular technology at a given time. What's in fashion may not have anything to do with your modernization-related goals.

Personal preference for a given technology

You should ensure that the project teams steer clear of selecting a tech stack fitting their personal preferences. This approach bears no fruit for your business.

Failure to make technological room for the future business growth

One of the consistently praised properties of well-developed modern technological solutions is their scalability. It makes sense to apply it to your case, especially if your new software is supposed to be flexible enough not to block your organization's growth anymore.

Having a UX and technical overview and being aware of these traps along the way, you may proceed to the software modeling stage.

KNOW WHERE YOU ARE GOING: MODELING STAGE

Once you determine the as-is state of the system, the modeling phase may begin. Developing a profitable digital product for a mature business is about creating value for the target user through technological upgrading. **A prerequisite for bringing benefits to your end-users is an in-depth understanding of their needs and paths through user scenarios**.

UI AND UX CONSIDERATIONS: LEARN ABOUT YOUR USERS AND THEIR PATHS

Multiple users in a modernization project have divergent needs that are realized depending on their paths within the system. For example, **the needs of occasional users may come nowhere near the expectations of a power user**, accustomed to UX intricacy and utilizing a more sophisticated interface. In much the same way, the IT operator may wish for a possibly utterly different set of functionalities when using your system.

These needs also determine how users navigate across your system's functionalities. Disregarding the expectations of individuals who use your system while proposing a new model may, sooner or later, result in mounting complaints and malfunction-related reviews. What's more, if the intended changes cover your company-internal solution as well, any alterations uninformed by the opinion of internal users may even slow down their daily work. For these reasons, **it's recommended to start modeling the system with end-users in mind**. It is compliant with the business goals of modernization set at the discovery stage of the project and reflects the idea of modernizing and growing – especially important for mature businesses. In other words, it is your users' needs that should serve as a starting point for reaching your KPIs and maximizing revenue and profit in the long run.

Having analyzed user expectations, we can transition to **user story mapping**. Tracing back to Jeff Patton's idea of depicting the user's backlog through a map, it gives a bird's-eye view of user actions, allowing the software team to design the UX with those in mind.

> The story mapping technique combines mind mapping and storytelling to decompose the story of what the user essentially does within your app.

TIME TO START MODELING YOUR SOFTWARE

The time has come to conceptualize the new flow with the use of process models. The aim here is to gain **in-depth knowledge about the domains** within a system, create a **domain model** and provide a **visual representation of the architecture**. What are the tools that facilitate the successful completion of this stage?

First of all, elements of the earlier-mentioned Event Storming are of use here. Udi Dahan refers to events as "occurrences in the domain". Modeling-phase Event Storming is one way of exploring system domains with a concentrated view towards a final software solution. As Alberto Brandolini points out in *Event Storming*, events carry meaning, reflect state alterations to which we can apply logical rules, and help indicate process hurdles.

> The modeling-phase Event Storming technique is used to explore a system's domains, giving input for your new software solution.

Once you know a domain well, you can work towards creating a model of it. Let's break it down in simpler terms. As Eric Evans notes in *Domain-Driven Design. Tackling Complexity in the Heart of Software*, the user utilizes software to access a thematic realm called a **domain**. In this regard, domains hold a richness of knowledge about specific topics of interest to the user, and software is a gateway to those. Models are, in turn, a means to tackle this knowledge complexity in software (and business). In theory, **Domain-Driven Design (DDD)** is thus a philosophy for complex software development concentrating on the **core domain** and utilizing the joint work of Domain Experts and software professionals, resulting in speaking a common **(ubiquitous) language** within a clearly Bounded Context. But what does it mean?

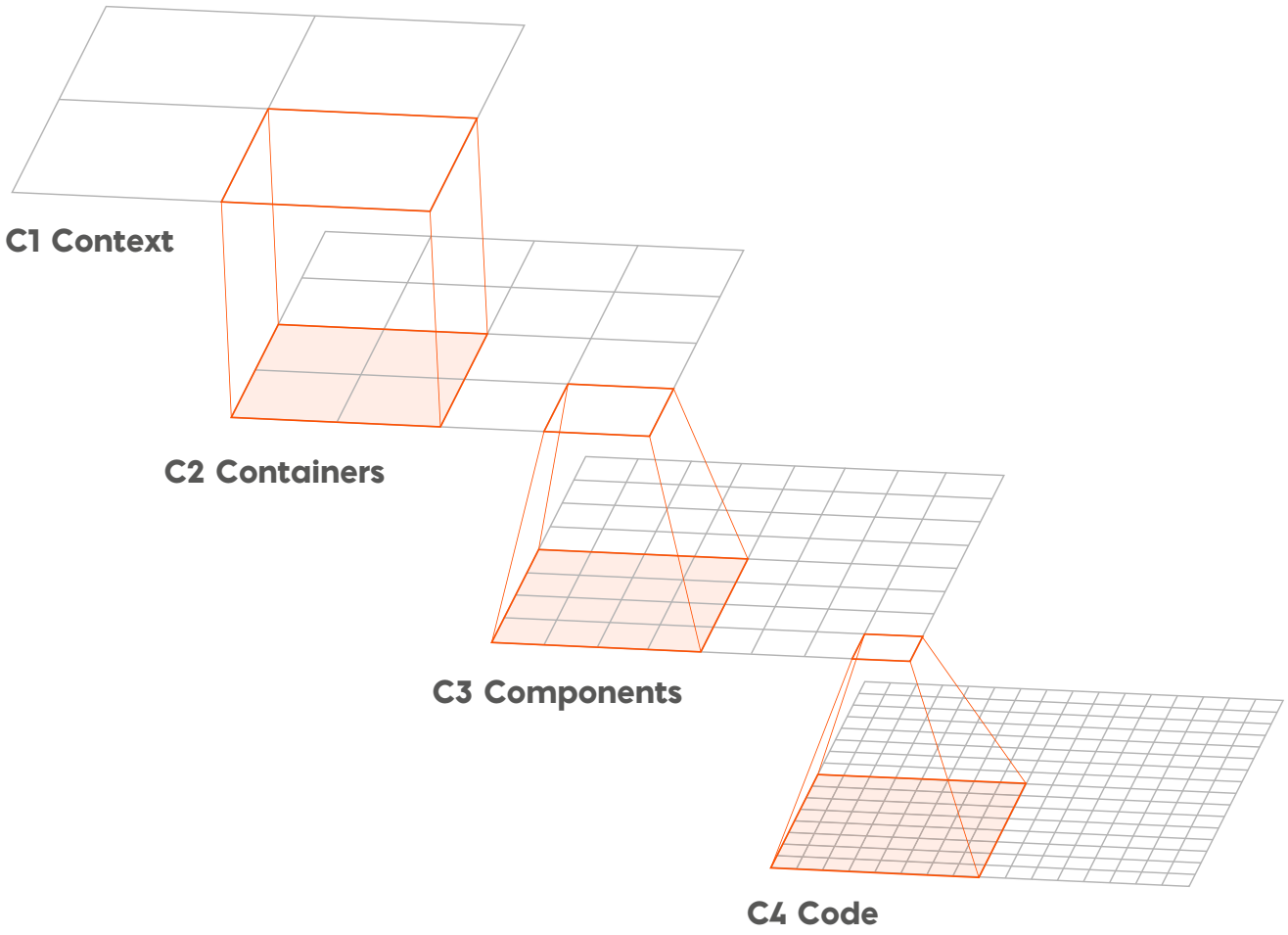
In the 2004 *Big Blue Book* mentioned earlier, Evans warns against using models unfit for a given domain. This idea of models befitting different thematic spheres brings us to the notion of **Bounded Contexts**. They constitute independent constructs made of domain-specific events and data-driven by rules, yet some of these partitioned contexts typically reveal a level of interrelatedness. Such **context mapping** is crucially important. Not only does it allow your tech team to **identify the interwoven rules-governed networks (Bounded Contexts) – or a lack of them – but also prevent system-internal entanglements** that you want to avoid by all means.

In short, DDD provides a solid organizational framework for conceptualizing different system processes and modules.

> The main benefit of using the Domain-Driven Design technique comes with the organizational structure it gives to the concept of your new system.

The very attempt to model your application using domains and Bounded Contexts will help render your product more well-organized. That is, in fact, the principal value of domain-driven exploration and development.

While DDD enables you to conceptualize the systemic processes properly, we should not forget about a proper projection of your system architecture. One of the methods used to represent the software architecture visually is the **C4 model**. It is advisable to use the C4 technique for projecting the architecture structure at various levels of detail. The C4 element in this standard denotes four levels of architectural complexity. The System Context (C1) part shows a kick-off point in that it encapsulates how a system is immersed in the surrounding environment. The C2 chart demonstrates the higher level with the system's constitutive bricks (containers). You can also have a more detailed look into a given container's structure – this is what the C3 level component diagram shows. Finally, C4 gives you an in-depth view detailing how a component is implemented.



Schematic visualization of the C4 model for system architecture

All in all, the C4 model enables you to get into the architectural intricacies in detail and provides a clear visualization of the system's skeleton. Now, having a good visual representation of the architecture, a thorough understanding of domains, and a conceptual model for them, you may move on to the planning stage.

JOURNEY ITINERARY: PLAN & PRIORITIZE

Parallel to determining the future shape of the upgraded product and right into transitioning to the planning stage, a modernization strategy is typically selected, not as the basis of the action plan but instead as its crucial element serving the software upgrade must-haves. In other words, **an approach to modernization should always constitute the best solution with respect to its goals**.

SELECT THE BEST ROUTE: SYSTEM MODERNIZATION STRATEGY

It is safe to say a major software revamp at the very least involves moving an app to the cloud, which, at this minimal level, is included in the Lift & Shift strategy. It involves slight disruption to the application and opens up a plethora of possibilities for further improvement, if applicable.

> Your software experts may apply **the Lift & Shift strategy** to enable a fast and smooth cloud adoption in itself and as the first phase of a more far-reaching modernization endeavor. We recommend performing such cloud enablement on the specific condition that the application is in a workable state.

For more advanced software needs, where the code necessarily needs to be changed to reach the upgrade's goals, we use the Augment & Refactor approach. **Your codebase may need refactoring if you cannot proceed with the development process or your system is underperforming**. Some prerequisites for a hindered development include outdated library versions, incomplete documentation, and weak unit tests. Performance issues, on the other hand, typically come out under heavy traffic. You should also not ignore a lengthy code alteration process that may make your code a candidate for restructuring.

> Choosing the **Augment & Refactor approach** is best when your system needs additional features, a performance boost, and significant changes to code necessary for cloud adoption. It can also be applied as an intermediary stage of a profound Complete Rewrite.

In terms of the number and qualitative scope of changes and modernization enablements, the **Complete Rewrite strategy constitutes the most wide-spanning and benefits-packed approach**. Necessitating a reformulation of business goals and budgetary assumptions, it is an optimal solution as per the new circumstances of your

organization. After a complete rewrite of the system, you end up with a future-proof solution with a new architecture built on a modern tech stack.

> Use **Complete Rewrite** if your business calls for a substantial and far-reaching upgrade in software.

All in all, the Lift & Shift, Augment & Refactor, and Complete Rewrite strategies should give you an idea of your organization's specific road to digital regeneration. As soon as you have this awareness of just how deep down into the system you need to go in your modernization efforts, a proper game plan should be developed.

At the end of this phase, the collaborative effort between your project team and software partner results in a product roadmap together with an additional list of prioritized objectives, project requirements, and the scope of work, as well as the timeline and cost estimation. How to make the best of planning then?

DETERMINE THE TRANSITION REQUIREMENTS

The first step is to decide on the transition requirements for your project. What has to happen to get specific elements and the entire system to work? Helen Huntley, an Analyst for Gartner, points out that "sourcing managers often focus on evaluating and negotiating contracts, paying less attention to how services will be transitioned". The truth is that you are less likely to reach your modernization goals and satisfy stakeholders if you fail to consider the project prerequisites at the pre-kick-off stage.

According to the CIO Magazine, **inadequate handling of requirements is the reason why 71% of unsuccessful software endeavors are, in fact, failures**, as demonstrated in the example of a sizable European company's call center management application:

> A few years back, a sizable European call center market player was planning to create an application managing their contact center processes. The initial software requirements agenda – created by 40 stakeholders – was 3,000-page long, creating a risky prospect of contact center consolidation postponement by four to five years. Hugh Cumming, currently the ADP Employer Services Canada CIO, who then handled this project, had resolved to drastically reduce the number of stakeholders and the requirements checklist. The reassessment process took him less than two months and involved a controlled collaboration with the major stakeholders. In the end, there were only five top stakeholders, and 90% of the requirements had been removed from the original list, leaving more room for the true must-haves. The final result was that the company implemented the project in 12 sites across the globe. The project's leader Hugh Cumming stresses the importance of a small number of stakeholders and a streamlined requirements list in their endeavor.

The above case study shows **the importance of the carefully chosen stakeholders pool and requirements' adequacy and priority** in streamlining the project's scope, timeline, and geographical reach. Project requirements are, however, relative to an individual business case. They may cover a wide array of cross-disciplinary aspects, including but not limited to: team upskilling, business operations and continuity requirements, regulatory and law-related aspects, security must-haves, and app support prerequisites.

ESTABLISH METRICS FOR MEASURING CHANGE EFFECTIVENESS

The second step is about establishing the exact metrics to which your project team will measure up the transition. These are relative to business objectives but may cover:

- *user-related feedback for the app, as measured by a fewer number of complaints,*
- *lost time level, related to the time lost on an underperforming user path,*
- *maintenance cost of the app post-modernization,*
- *the number of users utilizing the modernized app,*
- *customer lifetime value (LTV) related to users subscriptions and purchases,*
- *percentage of successful testing rounds determining the quality of the digital product,*
- *percentual QA code coverage of the key system modules and critical paths,*
- *performance measured by uptime percentage, time-between-failures, and time-to-recover,*
- *the amount of time necessary to fix a simple bug in the system,*
- *bug numbers per every sprint,*
- *level of Defect Removal Efficiency (DRE),*
- *level of readiness for third-party integrations and new features,*
- *security handling level, as determined by new library versions,*
- *application complexity level, measured by many decision points and their dependencies,*
- *cleaner code base, measured by a lower number of bugs and failures to the system,*
- *higher company value measured post-modernization.*

SET YOUR PRIORITIES THE RIGHT WAY

Once you know what the specific transition requirements are and how to measure the success of your modernization, you can move to prioritize the milestones ahead. One of the models that proved helpful with prioritizing the steps to digital enablement in our experience is the Guide to the Business Analysis Body of Knowledge® (in short, the BABOK® Guide). **It helps to set your priorities on the right course by providing a set of factors for categorizing the essential aspects to be put on your timeline**. The BABOK® Guide facilitates an investigation into various opportunities and related challenges. Exploring the notions of benefit, penalty, cost, risk, dependencies, time sensitivity, and stability, it helps to precisely define the priorities of your project.

> The advantage of using the BABOK® model is that it allows you to explore the opportunities and challenges within your timeline.

Another prioritization tool worth mentioning is the **MoSCoW technique**. It is a software development method enabling project participants to identify the obligatory and secondary-importance elements on their schedule. Within this practice, there are four prioritization categories: must-haves, should-haves, could-haves, and won't-haves. By identifying the must-haves and should-haves on your timetable, as well as assigning lesser importance to could-haves, your project participants get a clear picture of what requirements to focus on. To quote Jennifer Stapleton, **a system's minimal usable subset is created as a result of completing project must-haves**. Won't-haves, in turn, represent the redundant assignments on the current to-do list.

MoSCoW priority labels	Priority level	Disambiguation
Must haves	Indispensable	Requirements obligatory for the project's success (system fundamentals which determine its usefulness and usability)
Should haves	Important	Requirements that should be met but do not have a decisive influence on the system's usefulness or usability
Could haves	Desirable	Optional (non-necessary) requirements per given development period
Won't haves	Redundant for a given development period	Requirements that may need to be addressed in further development phases

You may want to pay attention to whether your software experts are able to produce similar priority scales for your specific requirements. Once you have those, together with the timeline, scope of work, and cost estimation, the development phase may begin.

Put modernization theory into development practice

With an improvement roadmap at hand, it's time to get your hands dirty with development. In the second part of this whitepaper, we invite you to think about the key elements of software modernization – architecture, infrastructure, QA, and delivery – in practical terms.

ARCHITECTURE

Let's say you wish to hold a lavish dinner at the end of the week. How would you start this quest: by rushing to the grocery store and buying random ingredients or by listing the dishes that you'd like to cook? We're not trying to say that software modernization is as easy as throwing a dinner party (which, by the way, can be a serious challenge) but rather show you that while it might be tempting to jump straight into selecting software development tools, frameworks, etc., it makes much more sense to start by adopting a broader perspective on your system. **This brings us to the complex topic of architecture, cleverly and accurately defined by Ralph Johnson as "the important stuff... whatever that is".**

THERE'S NO GOOD OR BAD ARCHITECTURE

Having built your digital product for a couple of years now, you already know that the most common answer to all software-related questions is "it depends" – and architecture is no different. When designing a complex system's architecture, you need to bear in mind a multitude of factors. On the one hand, there are the business goals as well as budget and time constraints. On the other, there's a nearly infinite list of technological characteristics, such as sustainability, fault tolerance, or autonomy, to name just a few.

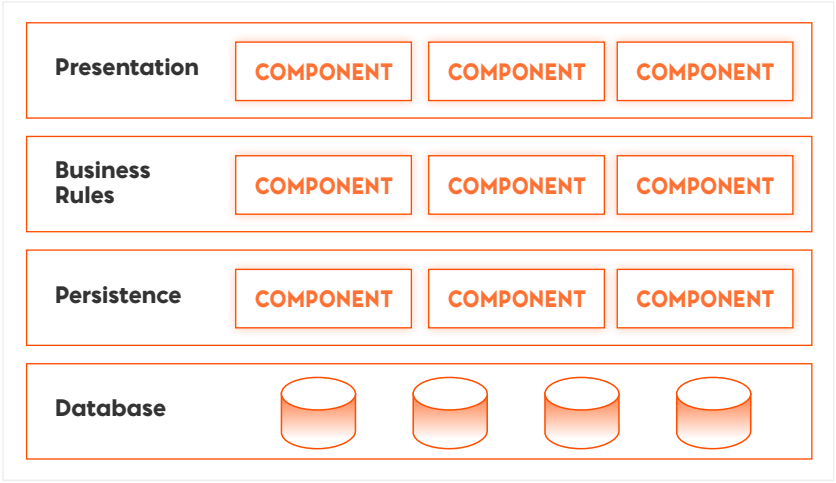
What's most challenging about the factors mentioned above is that they rarely go hand-in-hand. To give you an example, enhancing security can harm performance, whereas greater customizability means harder debugging. Therefore, as Mark Richards and Neal Ford observed, **we can't speak of right or wrong architectural decisions but rather of trade-offs**. It all depends on the context – in your case, software qualities of major importance to your organization and its goals (see what we've done here?).

ARCHITECTURAL STYLES AND PATTERNS

Even though there are no silver bullets in software development, there are architectural patterns that address certain needs more effectively than others. Let's now discuss the most popular ones, taking a closer look at how they can support – or hinder – modernization and growth.

Monolithic architecture

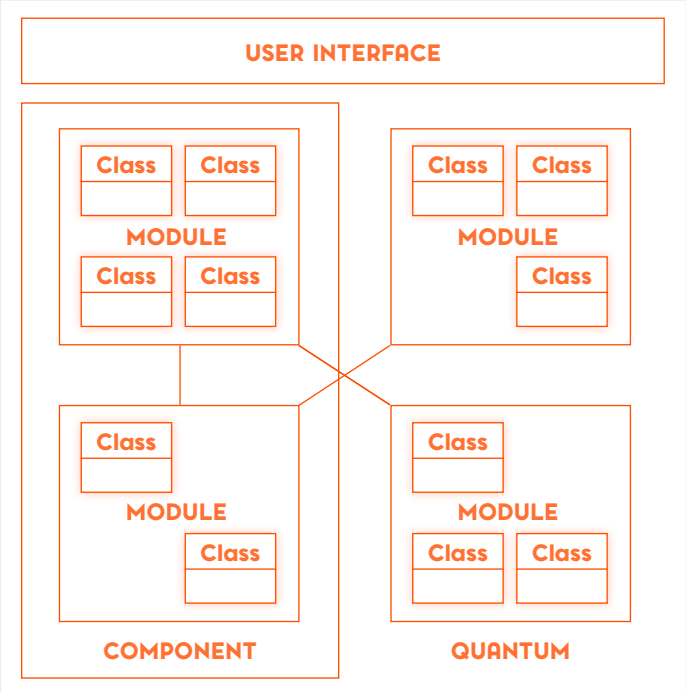
First comes a self-contained monolithic architecture, whereby all functionalities are deployed together. Associated with the old practices as well as susceptibility to coupling-related problems, this pattern gets a lot of bad rap in the IT world and is often a go-to candidate for modernization. Remember, though, that **the monolithic world extends way beyond the stiff unstructured architecture and encompasses some more beneficial patterns**, such as:



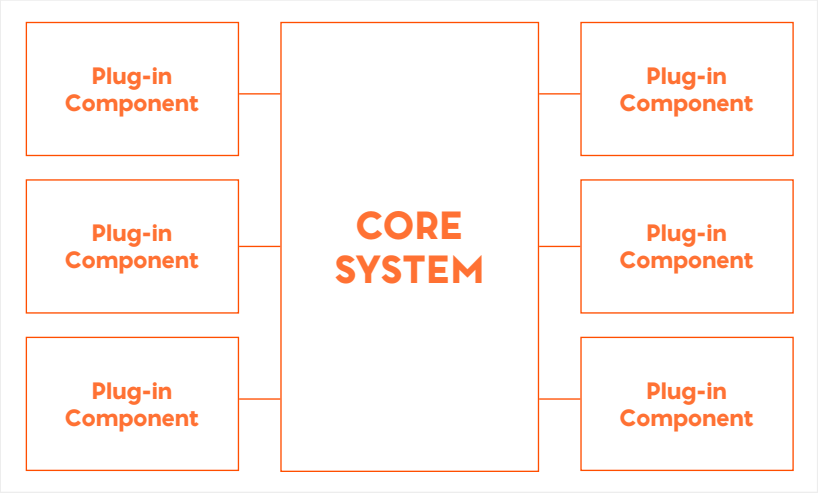
Layered monoliths whereby technical capabilities are grouped into layers. Characterized by high internal but low external coupling, this pattern is easy to understand and facilitates introducing changes to the system – although it may not offer the optimal performance or scalability.

Layered monolith architecture. Source: Building Evolutionary Architectures

Modular monoliths that profess isolation of and little dependency between respective modules. Keeping coupling at a minimum, modularized monoliths offer a practical alternative to microservices – especially if you're anything like Shopify and don't want to be bothered by maintaining multiple deployment pipelines or increased latency.



Modular monolith architecture.
Source: Building Evolutionary Architectures



Microkernel architecture. Source: Software Architecture Patterns

Microkernel architecture made of a monolithic core and independently deployable, plug-in modules hooked into it. While the former contains the general business logic, the latter account for custom enhancements and modifications. To give you an example, microkernel architecture can offer improved performance at the cost of scalability – but here you can read more about the practical advantages of microkernel on an example of a claim processing software.

Event-driven architecture (EDA)

Then, we move on to EDA, which is "made up of highly decoupled, single-purpose event processing components that asynchronously receive and process events". Broadly speaking, event-driven architecture is a way to go if you're concerned with deployment, scalability, and performance.

We can distinguish two implementations of this pattern: broker and mediator. The former is suitable for simple event processing flows and encourages making evolutionary changes to the system. The latter works well with multi-step events and, in comparison to broker topology, it's prone to increased coupling.

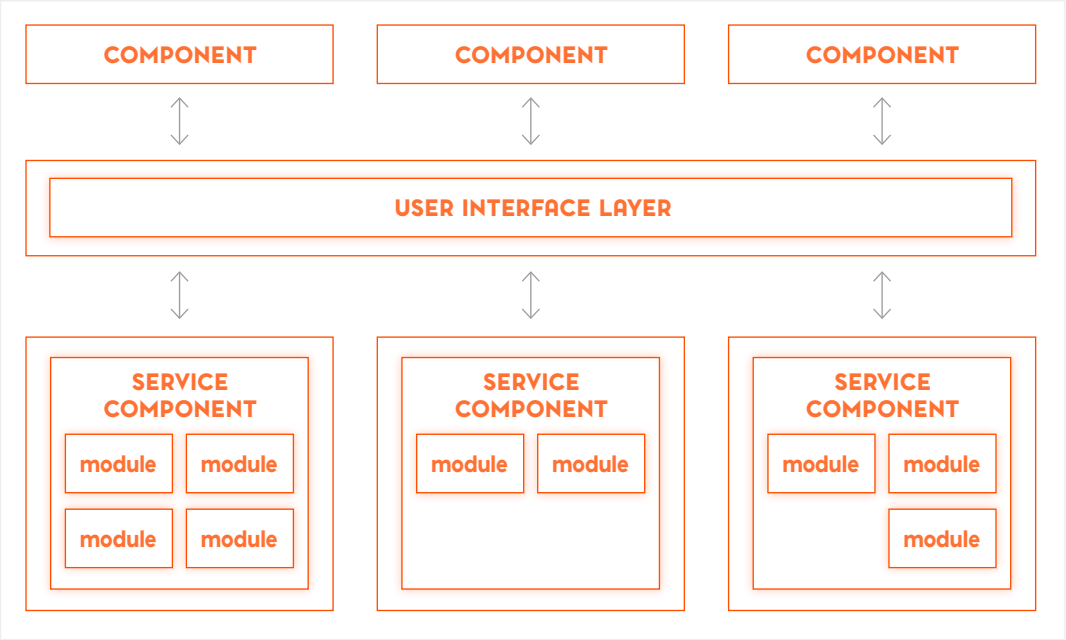
Service-oriented architecture (SOA) and microservices

Next comes SOA, defined as a "loosely-coupled architecture designed to meet the business needs of the organization". In this case, the application is made of independently managed and deployed components (also referred to as services) with clearly defined boundaries between them. As each service performs a certain business function, this pattern on the one hand has the potential to foster continuous improvement but on the other hand, calls for thorough planning and architectural discipline.

If the "independent deployment" rings the microservice bell, you're absolutely right! SOA and microservices are similar in many respects, e.g., pushing for loose coupling and reusability. The main difference is the scope: enterprise-wide adoption for the former and application-wide adoption for the latter.

Microservice architecture is based on seven tenets, which include shaping service boundaries along business domains, embracing automation with CI/CD, or drastically decreasing coupling – even if it is to result in duplication. As such, microservices offer a wide range of opportunities, from improved scalability and flexibility, through

improved team autonomy, to a more diverse tech stack and reduced time to market. However, these benefits come at a price of potential complexity or even confusion – not to mention that the rapid and premature adoption of microservices can do more harm than good.



Microkernel architecture. Source: Software Architecture Patterns

Serverless architecture

Last but certainly not least comes serverless architecture whereby the application – to some extent contradictory to what the name suggests – still runs on the servers, but the ones managed by cloud providers, such as Microsoft, Amazon, or Google. This pattern can be realized as **Backend as a Service (BaaS)** or **Function as a Service (FaaS)**. Serverless architecture can free project teams from worrying about the infrastructure (which we'll discuss in more detail shortly), cut expenses, and facilitate scalability. At the same time, however, it comes at a threat of vendor lock-in and unfavorable billing terms. As we've said earlier, there are always trade-offs.

> We mentioned that there are no right or wrong architectural decisions – only those that do or don't match your goals. However, there are also antipatterns: choices that seem beneficial at first glance but upon careful examination turn out to be a big mistake. Some commonly-discussed antipatterns are **the big ball of mud** and **the last 10% trap**. You can read more on that topic here.

EMBRACING THE (INCREMENTAL) CHANGE

As you're reading this whitepaper, it means that your company has evolved – and so should the architecture of your system. It would be great if excellence equaled longevity, but that's simply not the case in software development. Think about eBay and how the e-commerce giant tore down their system a couple of times – not because the software written in C++ or Java was bad in itself but because the context changed.

As far as the tempo of your system's modernization is concerned, it again depends. Maybe you have in mind the often-desired path from monolith to microservices – but trust us, it's certainly neither the only nor the always best option. **Instead of defining targets and jumping straight into the "how" of designing architecture anew, reflect on the "why"**. Think about the current state of your product, your business goals, all aspects of the software that will be affected by the upcoming change. As we said before, there are no architectural silver bullets, but **more often than not, it turns out that the incremental change is what you need**.

SUPPORTING BUSINESS CAUSE WITH ARCHITECTURE

- *Don't be afraid to make considerable changes in the architecture when an equally significant transformation in the surrounding environment happens. The only constant in life – and business – is change, so processes and tools should follow it.*

- *Automate everything because one thing computers are undoubtedly great at is doing the same thing over and over again. Maybe spending 3 hours on automating a 10-minute task seems unnecessary at first glance, but if it's a daily job, you'll notice the return on investment in a month – and you'll gain additional 10 minutes of life.*

- *Plan maintenance because even brand new products need small tweaks every once in a while. Like cars, applications require continuous care to be at their best at all times. Rust starts small but, when ignored, it can eat up the whole car. Don't think of it as a cost but an investment into the future.*

- *Measure what you can. Technical or business measurements will give you insight into how well your application is doing. You will get early notice when things start to go sideways.*

INFRASTRUCTURE

Imagine your application is struggling with high traffic or a special event. Then, think of getting complaints from customers about not being able to access some features or, in the worst-case scenario, the whole system. Sounds like a nightmare come true, doesn't it? **Well, these are the signs that you might want to take a look at your software's infrastructure.**

COMMON INFRASTRUCTURE PAINS

Every client who seeks software modernization has different infrastructure-related concerns. Some want to follow the expansion of their market, while others feel like their current infrastructure limits business growth. Whichever you are, let us make a valid generalization: **most outdated applications work on dedicated servers**, be it a local hosting company or an EC2 instance in AWS, treated as yet another server. Usually, **such a setup is struggling with natural business growth**.

How about a practical example? Imagine a situation where a server is placed inside an office building. Suddenly, a power outage that spans the entire district happens, and your application is unavailable for an extended time. Or maybe your system grows to the point that the server you used can't keep up with the current load – it's getting slow, or traffic is so big that some users simply can't access the page anymore. The point we're trying to make is that **when planning infrastructure, you need to think not only about the present situation but also about how the application will behave when your business grows to the point of exceeding expectations**. This doesn't mean you need to pay from day one for something you think might happen in ten years. However, the system's infrastructure should be prepared for scaling.

It's also vital to discover bottlenecks and think about potential disaster scenarios, e.g., what happens if the site goes down? Some businesses are ok with little downtime; others need to take extra steps to ensure that there are ways to access applications in the event of failure. **And as a responsible CIO or CTO, you have to expect things to go wrong** – just as it happened with the OVH fire. That's why going with a cloud provider is a good option. Companies like AWS prepare for such situations, storing data in strategically placed data centers to minimize risk.

CHOOSING THE RIGHT INFRASTRUCTURE

When you think about migration to the cloud, it might be tempting to follow in Netflix or Spotify's footsteps and embrace microservices. This, however, can be a double-edged sword. Think of microservices as separate applications that talk to each other and have different teams working together. Going for this approach means introducing major changes in the application itself and requires a lot of investment in terms of development time. It can pay off long-term, but several months may pass before you get to reap the benefits. **Generally, at the onset of software**

modernization, it's a good idea to solve a few bottlenecks of the current setup first and, if possible, prepare infrastructure for future expansion.

Speaking of designing infrastructure, it's also advisable to think about how well it can be maintained. As time goes by, your team is likely to change – but you don't want to put the business at risk only because the recently-hired devs can't get their head around system infrastructure, do you? To prevent that from happening, **we'd recommend implementing Infrastructure as Code (IaC), whereby your infrastructure configuration is stored in code repositories, just like your application**. IaC gives you greater control over what's running and allows you to create new environments mirroring your existing ones easily.

MODERNIZATION AND PATHS TO THE CLOUD

The easiest migration type is Lift & Shift. In this case, **application changes are minimal and made only so that the software can work in a new cloud environment**. The majority of work happens on the infrastructure level. Such a scenario assumes the use of generic cloud services that can be tailored for scale-up right away. During preparation, the development of your system remains undisturbed. When new infrastructure is ready for migration, it can be carried out with minimum downtime – just enough to migrate databases and avoid data desynchronization. When going with Lift & Shift, additional steps need to be taken, such as rollback scenarios and the following cleanup of old infrastructure.

Cloud optimized approach assumes we are already on the cloud, having clean Infrastructure as Code. The first step, in this case, is taking a closer look at the core services of your application. Cloud providers offer many solutions optimized for specific goals, ranging from an auto-scaling database, machine learning solutions, or media converters. You can leverage them to speed up the development time of new features. It's also worth mentioning that such solutions offer by default monitoring and automatic backups. They can also be set up in high availability mode that allows for auto-recovery from a failure.

Cloud native, on the other hand, applies to advanced projects, where the correct use of infrastructure is critical to business success. Speaking from our experience, in this scenario, we collaborate with clients that go full serverless, working on non-relational databases and having their APIs on FaaS solutions. This allows them to focus entirely on features, while infrastructure is built to scale when needed. Nonetheless, most companies don't need to go that far. While it seems like a silver bullet, the cloud native approach comes with increased costs – since cloud providers shoulder responsibility, you need to pay for their efforts.

MIGRATION TO THE CLOUD IN PRACTICE

Let's now discuss the tools and processes that facilitate migration to the cloud. To do so, we're going to refer to our own experience. When a tech partner migrates a client to a cloud provider or takes over an existing cloud setup, they want to make sure it will be easily maintained and expanded. At Merixstudio, **we create an IaC using Terraform**,

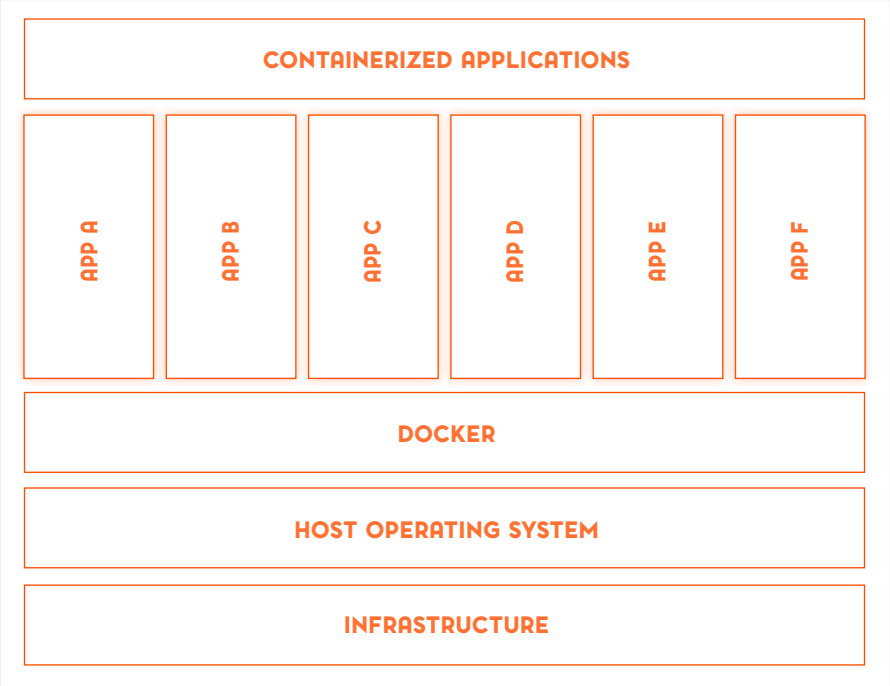
which allows us to easily control our environments, deploy new ones, add or delete resources, and mirror such changes on all of them.

We're also big fans of containers and are used to creating applications as docker containers. Doing so enables us to deploy applications in fully scalable solutions, e.g., a Kubernetes cluster or an AWS ECS (Elastic Container Service). This way, we don't have to worry about library updates or maintaining security updates of servers themselves. Instead, we can leverage auto-updates from cloud providers. We can also set up CI/CD to do rolling updates of new versions of an application, thus having no downtime on releases.

CONTAINERS

Most legacy applications run their services directly on servers, which means each running service can potentially impact others by locking specific ports or even modifying shared files. When unintentional, it can result in the application going down. Some services don't use all resources available, thus increasing the cost of infrastructure at scale (as N services require N servers). Therefore, when you modernize an application, it's a good idea to go for containers.

In general, unless you already aim at specific cloud solutions, **using containers gives you the flexibility of choosing infrastructure that containers will run on and better utilize resources**. A good starting point for introducing this solution to your app is taking a closer look at the local development environment. If you're using a framework, there is a good chance you'll find an example of docker image that already works with it. If the application is fully custom, such a container image would be built to contain services required to run the application.



Containerization with Docker. Source: Docker.com

OBSERVABILITY

As an owner of an outdated application, you might be lucky enough to have a few standard statistics of your server: uptime, CPU usage, etc. The bad news is, it's not enough. **Good monitoring and alerting setup goes hand in hand**

with business growth. When choosing cloud providers, you can get many metrics and alerts by default, e.g., AWS offers Cloudwatch as a center of your resources' metrics. The data provided can be used not only to create graphs and dashboards but also to alert on issues and trigger scaling of resources. A good example is ECS, where more EC2 instances (running containers) can be added if CPU or memory metrics reach certain thresholds. You can also monitor the application itself, using APM tools (Sentry, NewRelic) to observe the behavior of the system, discover bottlenecks and receive logs or traces from the application code.

BULLETPROOFING SOFTWARE WITH INFRASTRUCTURE

- *Cloud providers help to keep web applications online and easy to control, so you can focus on business growth instead.*
- *Rome wasn't built in a day and neither was any complex system. Bearing that in mind, plan in advance to scale the application from an infrastructure perspective.*
- *Containerize! Just like application built-in modules, infrastructure also scales faster and better when using containers.*
- *Remember to monitor your application. This way, even if you're dealing with complex setups and multiple computing machines, you can quickly react and ensure that your software remains stable and secure.*

QUALITY ASSURANCE

Imagine you're cooking an exquisite Sunday dinner we've mentioned at the beginning of this chapter. How do you make sure it turns out delicious? First, you choose the right ingredients – and second, you taste the dish throughout its various stages to ensure that it's neither too bland nor too spicy. After all, you wouldn't like to spoil what's tasted well so for, would you? Software modernization ought to follow a similar path whereby you start by choosing the right solutions and proceed to not only implement them but also test how they work out. **That's where Quality Assurance – an inseparable part of the effective modernization process – steps in.**

DRAFTING THE TESTING STRATEGY

The first step to ensure the high quality of the revamped product and the resulting impeccable user experience is the careful examination of the software's current state, and the QA practices implemented so far. As part of the initial QA audit of your soon-to-be-revamped system, it's worth taking a closer look at:

- **the documentation** – the more detailed the specification and the description of business goals or target group, the smoother the development and testing process;
- **levels of software testing covered in the project** – it might turn out that the lower levels of software testing, e.g., unit tests or integration tests have been neglected, and E2E tests have been prioritized;
- **automated vs. manual tests ratio** – while there are certain cases where manual testing is the way to go (e.g., identifying non-obvious issues with exploratory testing), a wide range of tests can and should be automated to make software modernization more efficient;
- **the current and future state of the tests** – as the system grows and changes, so should the pool of test cases, which is why test maintenance is key to successful modernization from the very beginning.

Taking these things into consideration allows drafting a comprehensive Quality Assurance strategy and aligning it with the development process.

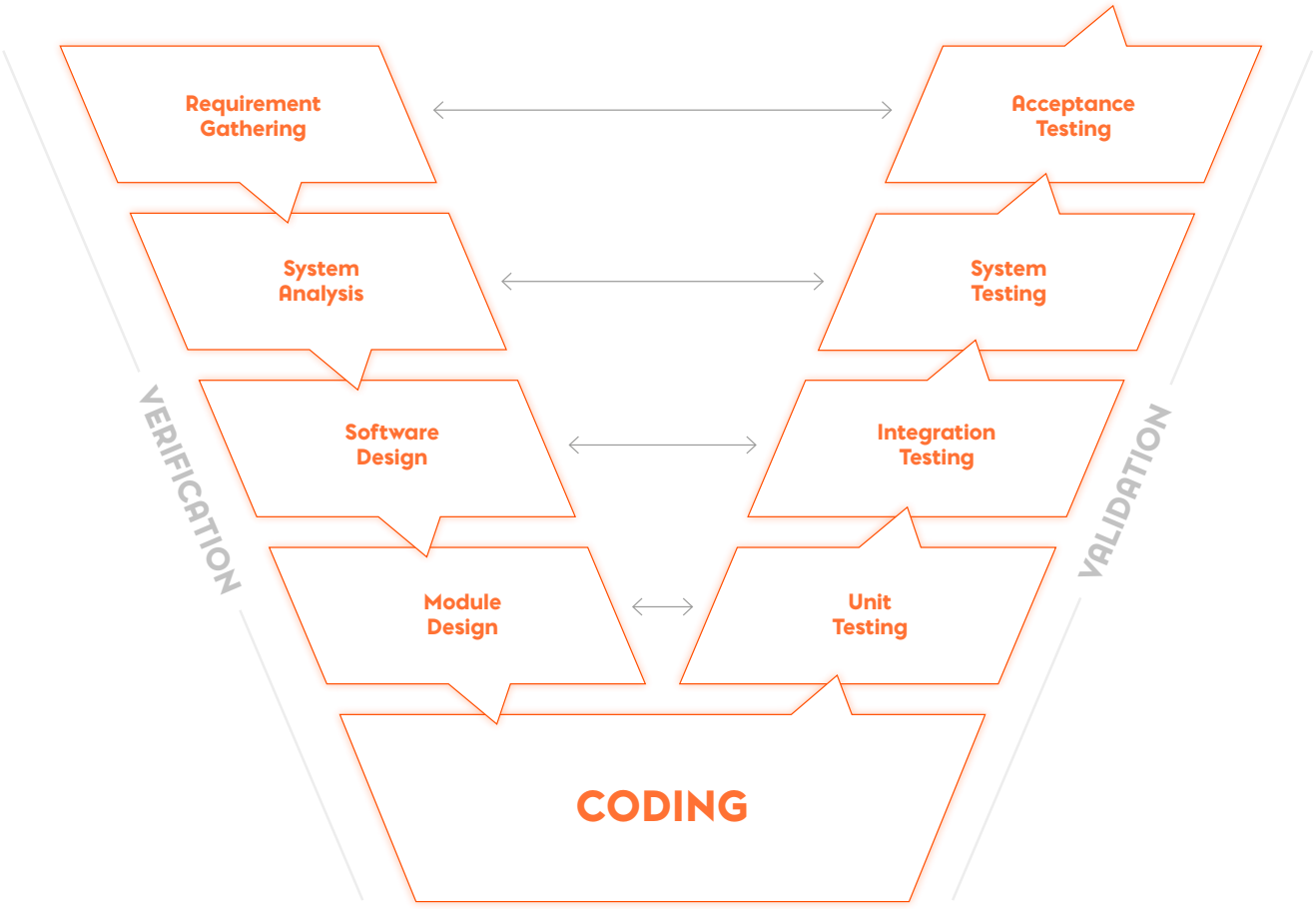
KEY QA CONCEPTS FOSTERING MODERNIZATION

Before we dig into the modernization-specific practices, let us refer to some core testing concepts applicable regardless of the project type in question.

Levels of testing

When thinking about software, one can distinguish several levels of complexity, starting from small components, through integrated modules, to the complete complex system. As proposed by the V-Model, for each development stage, there's a corresponding testing phase:

- **unit tests** – preoccupied with individual components fulfilling the requirement, relatively cheap to automate, and usually executed by developers;
- **integration tests** – checking the flow of data between software's components;
- **system test** – aiming to discover how all components will work together;
- **acceptance tests** – verifying whether the system meets business requirements and goals or not.

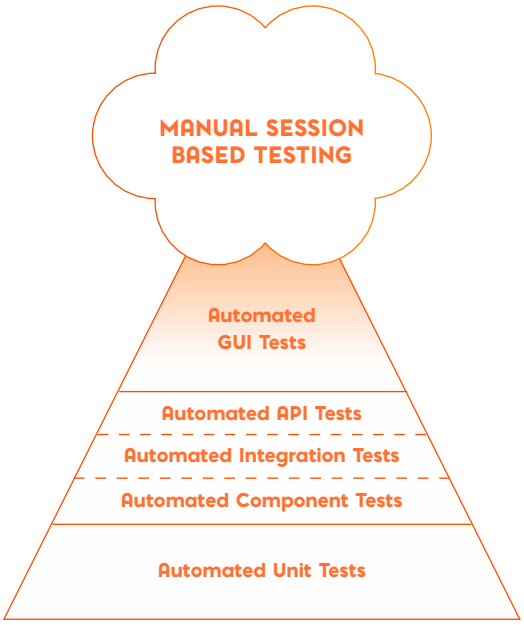


V-Model

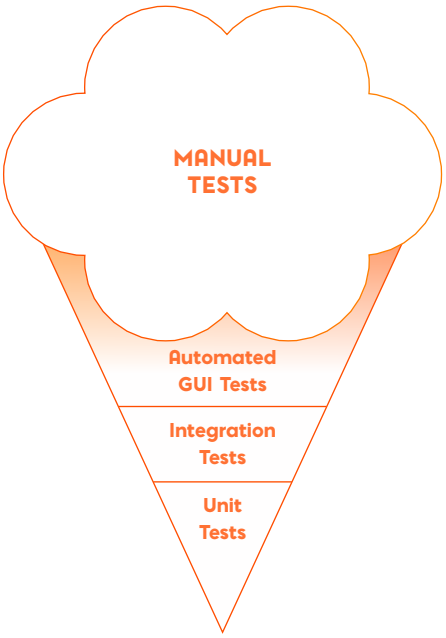
The unit to acceptance testing order is a matter of not only complexity but also the efficiency of QA practices – and when you need to both grow and modernize, neither level should be overlooked.

Test automation pyramid

Another QA concept worth referencing here is test automation. If your goal is to deliver high-quality software faster (which is most likely the case if you're about to modernize and grow at the same time), continuous delivery is the answer. With so much going on in the pipeline, however, **the only way not to get overwhelmed with time-consuming and repetitive testing tasks is to automate them** – and the test automation pyramid, which helps determine the types of tests that should be included in an automated test suite as well as outlines their sequence and frequency, is there to streamline the process.



Ideal Software Testing Pyramid



Software Testing Ice-Cream Cone Anti-Pattern

And while we could go on about the pyramid for pages, let us just quote Martin Fowler on the key outtakes: write tests with different granularity, and the more high-level you get, the fewer tests you should have.

API VS. E2E TESTS

Marrying the two topics discussed above in the context of modernization, let's take a closer look at **API tests, a part of integration testing that helps in maintaining the functionality and logic of the application**. On a technical note, good communication within the team is key when it comes to API testing. Developers should ensure that testers can quickly validate APIs endpoints. An example of such collaboration would be API contracts, discussed and easily understood by both parties; OpenAPI Specification may be documentation for it.

The key thing is that in comparison to automated E2E tests, which are expensive to perform and maintain, automated API tests can save both money and time by allowing spot potential threats for the project sooner. At the end of the day, however, you should remember that **API and E2E tests play a different part in a QA process, and as such, they're not contradictory but complementary**.

FUTURE-PROOF YOUR SOFTWARE WITH THE RIGHT TESTS

Now that we've covered different levels of testing let's proceed to discuss two types of tests that complement the software modernization process: regression and performance testing.

Regression testing

Ever heard of the butterfly effect? It's a theory proposing that "small changes in initial conditions can lead to large-scale and unpredictable variation in the future state of the system" and can shed light on why regression testing is a must in software modernization.

Whether you're refactoring or rewriting the application, it goes without saying that your goal is an improvement, both in terms of business objectives and user experience. At the same time, however, modules within a complex system are more or less interdependent. **This means that even the slightest change to the application can affect its other components, much like the butterfly affects the weather in the theory mentioned above**. Sometimes, before it gets the chance to enhance the software, the new functionality breaks the old ones – and the metaphorical tornado of bugs can break out.

Luckily, there's a way to identify the incompatibilities in the system when they're still no more harmful than a butterfly flapping its wings. **It's called regression testing, and its purpose is to ensure that whenever a change is made to the code, the software performance remains impeccable**. This practice calls for close collaboration between QA specialists and developers, who inform one another about the possible issues induced by updates and new features. This cooperation is also about QAs spotting the part of the software that doesn't work correctly and devs diving into the code to fix it.

Regression testing is a continuous, at times tedious task that calls for constant widening of the test cases pool. As such, it is a perfect candidate for automation, which can pay off in **increasing the QA team's capacity, immediate feedback, and easier maintenance**. An example of a tool that can help with that, as well as test management in general, is XRay. That being said, you must remember that automation works best when it's supported by manual exploratory testing, which allows spotting the unknown and unexpected pitfalls.

Performance testing

Imagine enhancing the existing system with new functionalities or modifying the old ones only to find out that they put a strain on the architecture or infrastructure and ultimately make the load times unbearable. Simply put, the system turns from "good enough but needing an extra touch" to "driving users insane". It's one of the reasons why performance testing and software modernization make a perfect match.

Ideally, performance testing should be introduced early on during your modernization journey and begin with asking the right questions, such as what is the desired performance or which bottlenecks are deal-breakers for the end-users. This helps clarify expectations and set priorities straight.

The next step is for the team to delve into the development and monitor how the modifications influence the performance. The most common ways to do the latter are to perform **load and stress testing, which shows how the application behaves during normal to high load and aims to simulate the system's behavior under an extremely high load and the subsequent recovery**. Much like other QA practices, to be most effective and uncover the source

of trouble, performance testing calls for close cooperation of the entire project team, including developers, DevOps engineers, QA, and UX specialists. We told you that software modernization is a team sport, didn't we?

MAKING THE MOST OF QUALITY ASSURANCE

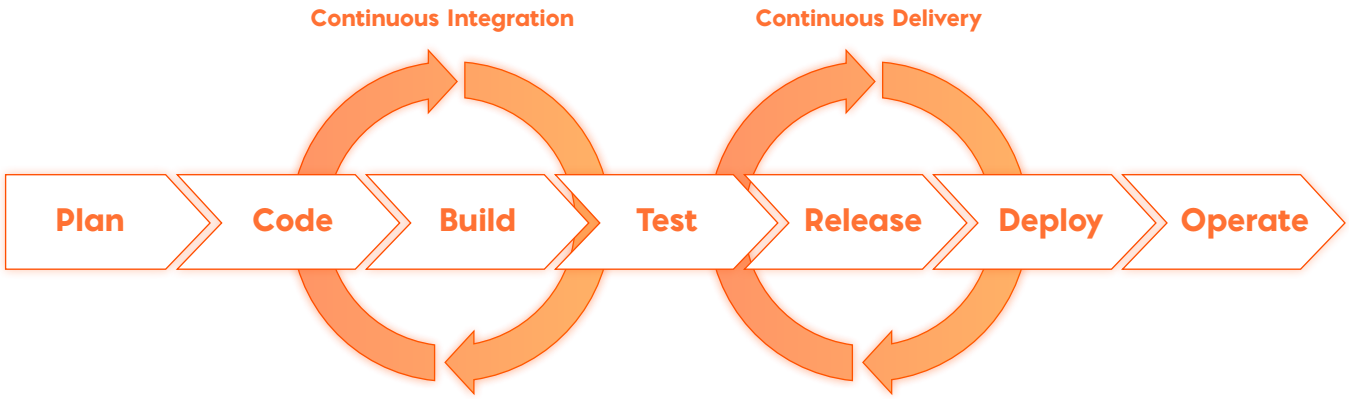
- *Provide the testing team with a detailed project description and comprehensive requirements.*
- *Trust your tech partner's team and give the QA specialists the freedom to choose tools and techniques that best suit the project.*
- *Don't underestimate the importance of unit tests. Since they're smaller-scoped than E2E tests, they not only allow identifying failures on the lower levels of development but are also easier to maintain.*
- *Automate testing activities early on to increase QA efficiency and product quality.*

DELIVERY

We're now heading towards the end of this publication. Before you rush to modernize your system, however, let us tell you a thing or two about delivery in terms of both deployment and workflow.

CONTINUOUS DEPLOYMENT

Continuous deployment is an important practice to ensure application stability and availability. Many outdated applications struggle with feature releases. Most of the time the reason is that application deployment was made during a PoC or an early development phase. It might have been executed manually, covering only the necessary things, and then patched from time to time. After a while, you realize that a simple feature release requires several hours of setup and often a small downtime of the application.



CI/CD process. Source: ArrowHiTech

There are a few best practices to think of when setting up deployment. One of them is **using code repository CI/CD tools, such as Gitlab, Github, CodeDeploy, Code Build, or Bitbucket**. Doing so allows you to build applications, run tests, and deploy them in different environments on every code commit. This way, you can quickly verify if the code is good enough to be deployed. A proper CI/CD setup should only allow for deployment if previous steps are passing, be that different tests or syntax checks.

Another rule of thumb is to **assume that the main branch can be released at any time**. Use lower environments for development and feature testing; once it's on the main branch, it should be stable enough to be deployed at any point. This way, you can deploy more often, spend less time doing so, and focus on your business instead.

PROJECT MANAGEMENT

So far, we've considered strictly technological aspects of software development. Now, however, it's time to discuss workflows and processes, which play just as important role in software modernization.

Most common risks in modernization projects

Let's not beat about the bush: modernization projects are anything but easy. If they were, you wouldn't be reading this whitepaper in the first place, would you? As a CIO or CTO, you probably have in mind a dozen project-related risks – but so does the tech partner's team if you're up for collaboration. The key thing is to **join forces in the pre-development stage discussed in the previous chapter and engage in risk management early on**. This way, you'll identify the most urgent risks that could jeopardize the modernization venture and the best ways to address them.

We know very well that the challenges you will have to rise to are unique – but based on our experience, we believe there's also a strong chance you'll run into one (or more) of those:

- entrenched testing, review, and acceptance processes that hinder the quick implementation of new features and prolong time-to-market;
- high chance of decreasing morale of the development team (be it in-house or outsourced) fed up with paying back the tech debt of the system;
- communication challenges between the in-house and the outsourced teams.

We've picked these three for a reason: they all have to do with project management, both on your and the tech partner's side. Although we could name a few ways to mitigate these risks, let's focus on one pair to rule them all: Agile and Scrum.

Agile project management

The Waterfall model is a part of many enterprises' DNA. We don't mean that it's inherently wrong because this methodology can work well in certain projects. At the same time, modern software development calls for easily reversible experiments conducted under controlled conditions – and that's what **short iterations promoted by Agile methodology are best suited for**. Throughout software modernization, the said tests are often reinforced or undermined, which is also most effective in shorter iterations. To prove our point, let us quote the example of the Ohio Tax system.

> In 2008, the state of Ohio commissioned the integration of over twenty legacy systems. For four years, the integrator followed Waterfall methodology, which resulted in missed deadlines and little effects. As a result, a decision was made to switch to Agile. Fast forward to 2016, 14 of 23 tax types have been integrated, not to mention that the project was expanded to include a frontend system. The system's project director attributes the success of the modernization efforts to the Agile iterative approach.

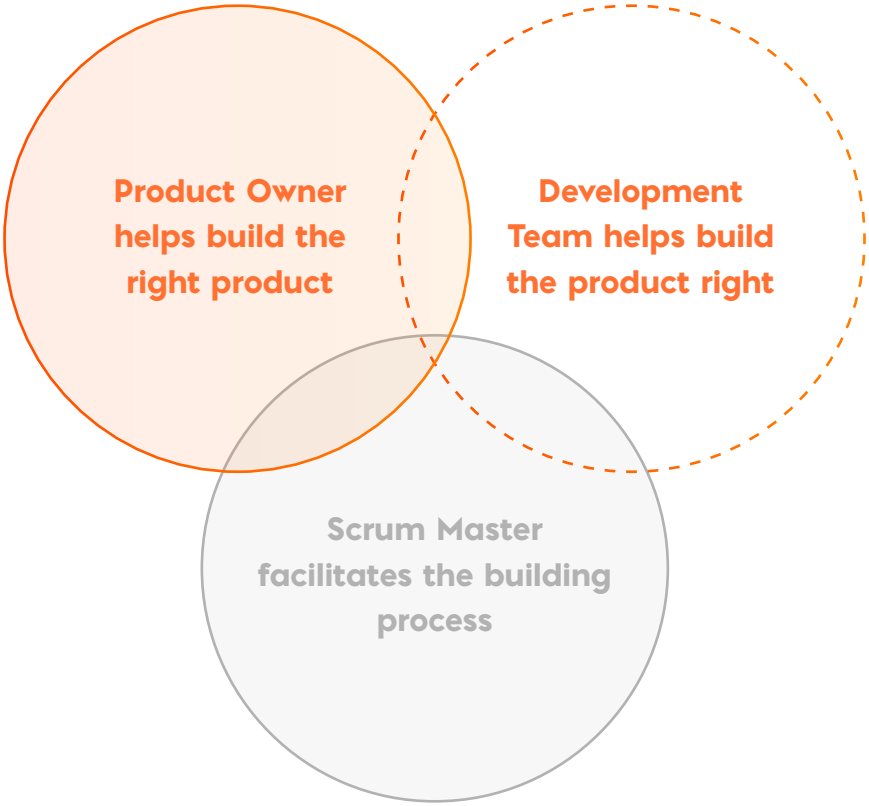
In this modernization case, turning from Waterfall to Agile resulted in small-scope iterations and smaller but quick and stable releases. Seems like we've just addressed the first risk, haven't we?

Scrum framework

Although Agile is not the ultimate answer to all problems, the other two risks mentioned above can be minimized by it as well – or, more precisely, by the agile framework called Scrum.

In the case of software modernization, it's only natural that nobody knows the existing system better than the people who've been working with and on it for the last couple of years. This is why modernization calls for a close collaboration between two parties:

- your team, including not only tech specialists but also a Product Owner, who knows the product vision, business value, and stakeholder's requirements;
- your tech partner's development squad and a Project Manager capable of acting as a Scrum Master.



Close collaboration in software modernization

The reason why we're focusing on the Scrum Master role is simple. Based on our experience, it's often the position missing in our clients' in-house teams and a role invaluable in facilitating the modernization workflow. Having direct contact with the development team, supporting their cause, measuring performance, and clearing impediments, **Scrum Masters have the opportunity to gently guide the team – both yours and your tech partner's – to the improvement of both efficiency and morale**.

FACILITATING MANAGEMENT OF MODERNIZATION PROJECTS

- *Give your PO the authority to make unquestionable product-related decisions and ensure they're truly in charge of the product backlog.*

- *When selecting a PO to cooperate with a tech partner, make sure they have deep domain knowledge, a proper understanding of the software that undergoes modernization, and good communication and interpersonal skills.*

- *Trust your tech partner in identifying threats and opportunities – having worked on dozens of projects from different industries, they can see your project from a wider perspective.*

- *If you feel that you could use support or simply another pair of eyes in organizing the project workflow, don't hesitate to make use of the tech partner's expertise; after all, your squad and your tech partner's teams work to a common goal.*

Don't let the outdated software slow down your growth

Match your big ideas with modern solutions that meet the users' expectations

Merixstudio drives digital innovation to businesses of tomorrow through end-to-end software solutions. Over the last twenty years, we've helped more than 250 clients bring their vision into reality. If you're on the lookout for a trusted modernization partner, why not get in touch with us?

contact@merixstudio.com

+48 570 001 928

ul. Małachowskiego 10, 61-129 Poznań

See our work Bēhance dribbble merixstudio

Check what our clients think about us Clutch

Follow our stories

Access more content on modernizing software