# 6 Approaches to Legacy System Modernization

Legacy systems have lots of drawbacks. They’re difficult to maintain, support, improve, or integrate with new systems. Their maintenance costs a lot—up to **$7 to $19+ million** per year if we’re talking about a large-scale app.

But it’s not easy to replace an old system. Especially when it handles vital business processes within an organization.

Legacy system modernization solves this problem by offering several approaches that won’t disrupt your internal processes.

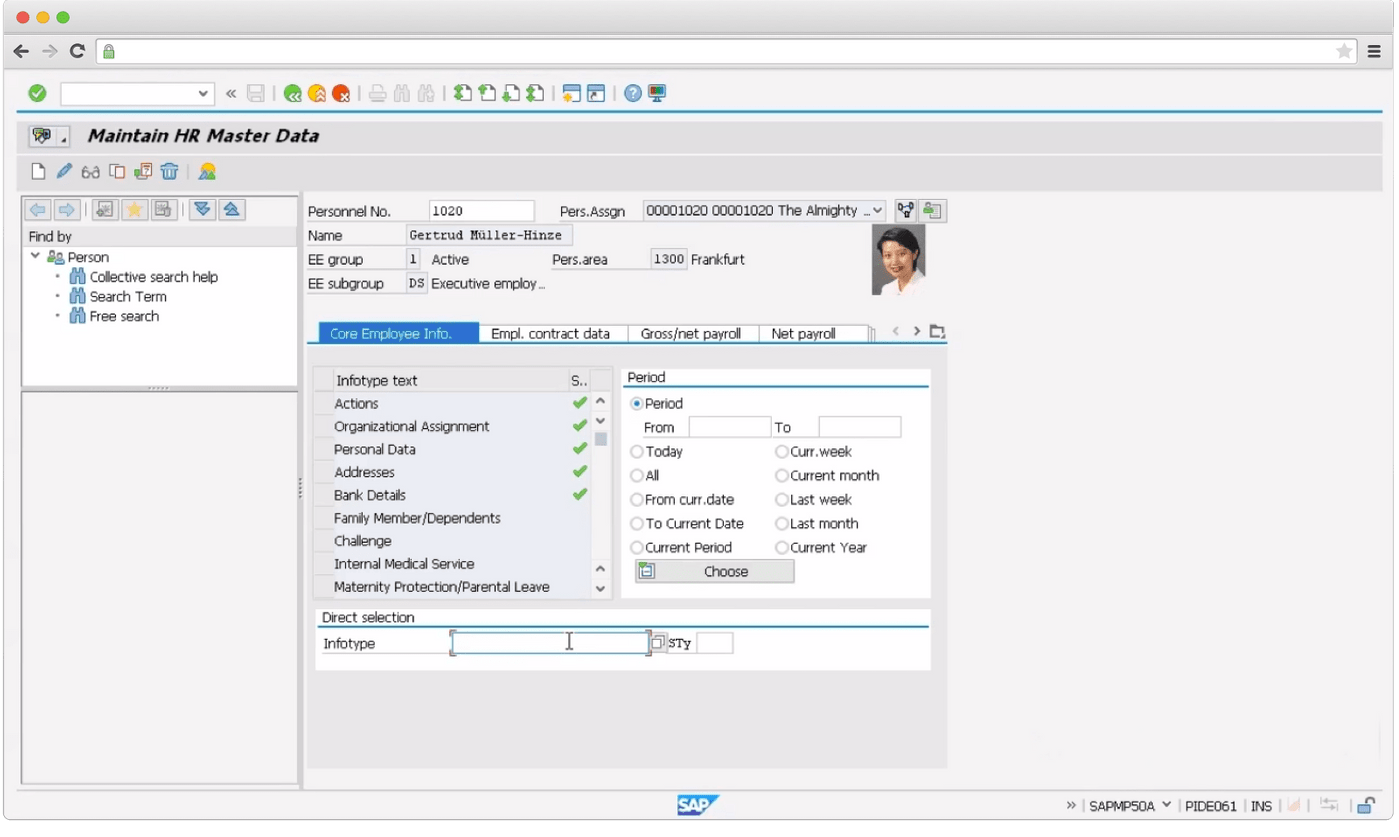
This guide reviews six common legacy system modernization strategies and lists the steps to follow before the modernization starts. Here’s what we’ll review:

**Table of contents**

* **[What Is Legacy System Modernization?](https://www.cleveroad.com/blog/legacy-system-modernization" \l "what-is-legacy-system-modernization)**
* **[How to Prepare for Legacy Systems Modernization](https://www.cleveroad.com/blog/legacy-system-modernization" \l "how-to-prepare-for-legacy-systems-modernization)**
  + **[#1. Define Your Goals for Modernization](https://www.cleveroad.com/blog/legacy-system-modernization" \l "1-define-your-goals-for-modernization)**
  + **[#2. Understand the Current State of the System](https://www.cleveroad.com/blog/legacy-system-modernization" \l "2-understand-the-current-state-of-the-system)**
  + **[#3. Choose a Modernization Strategy](https://www.cleveroad.com/blog/legacy-system-modernization" \l "3-choose-a-modernization-strategy)**
* **[Software Modernization Strategies](https://www.cleveroad.com/blog/legacy-system-modernization" \l "software-modernization-strategies)**
  + **[Replacing](https://www.cleveroad.com/blog/legacy-system-modernization" \l "replacing)**
  + **[Rehosting](https://www.cleveroad.com/blog/legacy-system-modernization" \l "rehosting)**
  + **[Replatforming](https://www.cleveroad.com/blog/legacy-system-modernization" \l "replatforming)**
  + **[Refactoring](https://www.cleveroad.com/blog/legacy-system-modernization" \l "refactoring)**
  + **[Re-Architecting](https://www.cleveroad.com/blog/legacy-system-modernization" \l "re-architecting)**
  + **[Rewriting (Rebuilding, Redesign)](https://www.cleveroad.com/blog/legacy-system-modernization" \l "rewriting-(rebuilding-redesign))**

**What Is Legacy System Modernization?**

When we call a system a legacy one, we usually think of corporate IT apps from the 90s or early 2000s. Probably written in COBOL or Fortran and having an outdated user interface like this one:



Human resource management module of SAP

But neither age nor programming language defines whether the system is outdated or not. 'Legacy' refers to any piece of software, technology, or system that slows downs or blocks processes in an organization, making it harder to adapt to the changing market dynamics.

Based on this, **legacy systems modernization means updating and replacing an outdated or inefficient system, processes and applications**—partially or fully.

Let's imagine you're an insurance company running an old ASP.NET application.

The app itself is meant to manage car insurance claims. Currently, it requires adjusters to use their digital cameras to take pictures, load them onto their PCs from SD cards. Then manually upload these pictures to the site.

Works if you have a couple of cases per day. And what if there are hundreds?

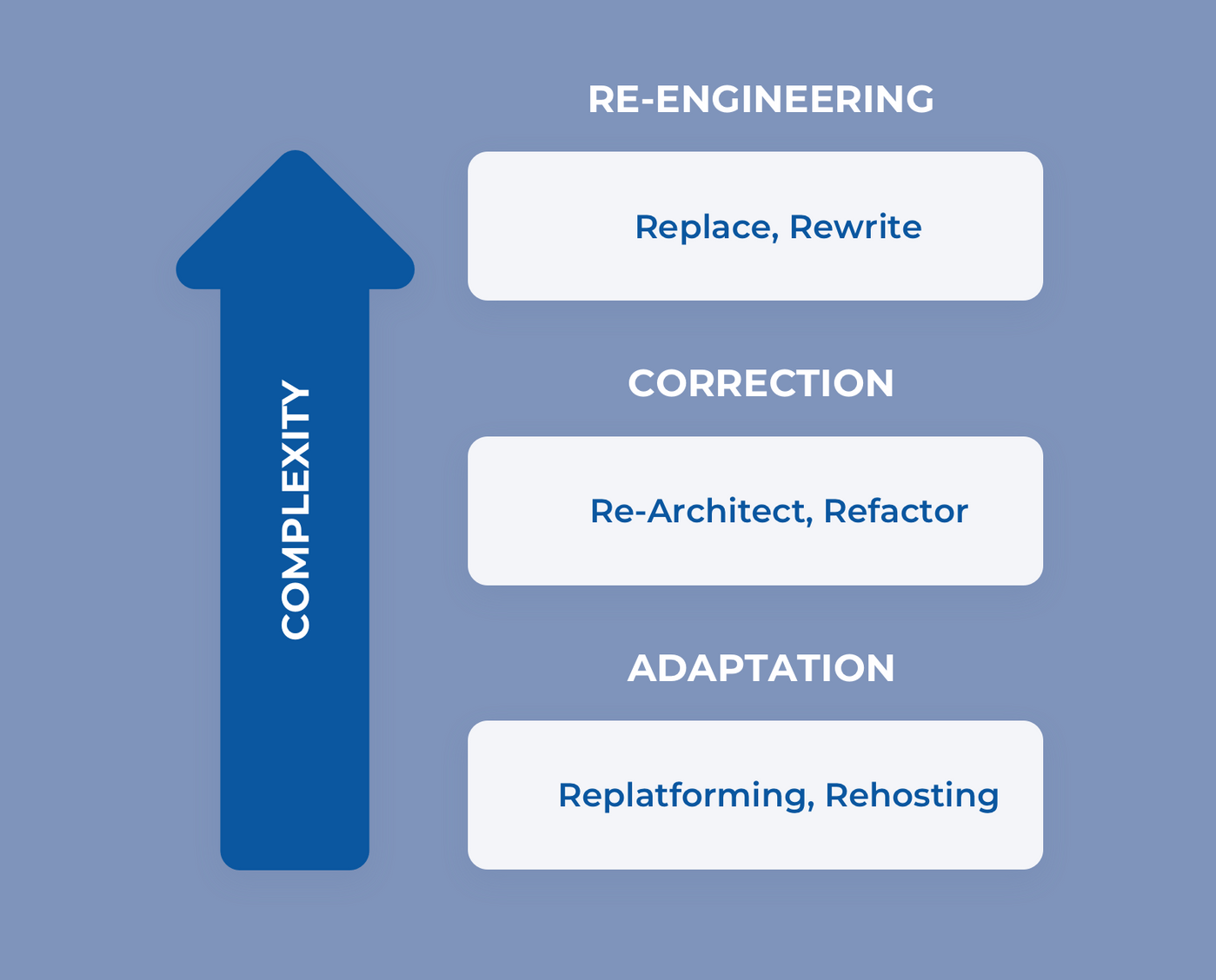
You may start by giving staff a mobile app, so they don't have to drag their laptops around. They start taking pictures, and the system automatically uploads them into Google Photos or cloud.

Or make a web application instead of a desktop one—more flexible, easy-to-scale, and accessible from mobile devices.

**How to Prepare for Legacy Systems Modernization**

Modernization does not always require a complete transformation of your IT stack. But what it does require is an honest assessment of where digital transformation is needed.

Before we review legacy software modernization approaches, let’s see what research and planning you need to do to prepare for it.

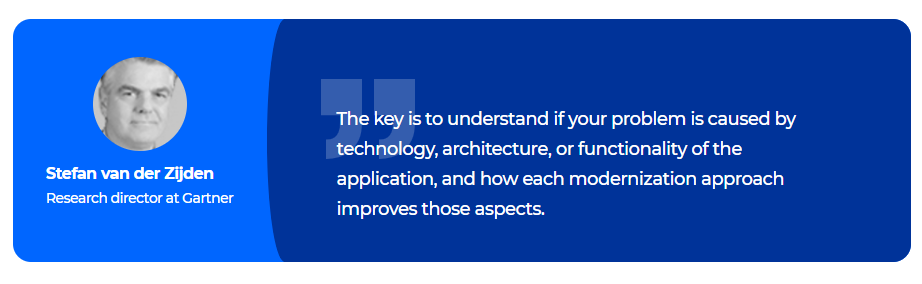


Modernization strategies by their complexity

**#1. Define Your Goals for Software Modernization**

Merely moving your COBOL/CICS environment to Java won't eliminate all the issues.

It's better to link modernization efforts to your business and tech goals—scalability, new features, faster time-to-market.



For example, modernizing a billing app and integrating it with a new e-commerce system could boost customer satisfaction and reduce service costs. That's because fewer customers need to contact the business about problems with their bills.

Depending on your industry, market, and goals, you can identify how legacy system modernization could impact the company's revenue, market share, improve customer experience, and many more.

**#2. Understand the Current State of the System**

The next thing is measuring how your current system performs. This will help you create the initial roadmap for modernization and prioritize the apps you need to modernize.

Start asking questions about your system’s **infrastructure**:

* How resources are being used?
* What chronic resource issues are there?

**Performance**:

* How is the software working now? What is its normal performance?
* Does the system throw any errors? Which ones?
* How is the end-user experience?

And its **impact on your business**:

* How does the system’s performance impact your revenue?
* How much time do staff members spend on the app?
* How does the app impact conversion rates or order value?
* Do errors or downtime impact customer service costs?

By having a full view of your system's performance, UI, infrastructure issues, and codebase, you and your team will be able to decide on system modernization strategy.

**#3. Choose a Modernization Strategy**

Now is time to use all the data you've gathered and choose a modernization strategy.

Of course, if you're working with a [**software development company**](https://www.cleveroad.com/services), they'll assess your system and suggest the best-fitting approach. Still, it's always better to get acquainted with all the options.

These are **six approaches** to legacy system modernization. But only three—**rehost, replatform, and refactor**—involve modernizing.

Let’s review all the approaches in more detail.

**Software Modernization Strategies**

Although rewriting or replacing may be the first thing that comes to mind, in fact, legacy software modernization should be done on a case by case basis.

challenge, the best approach depends on the problem you’re trying to solve. Replacement isn’t the only option.

That's why you need to do research and assess your current system—to decide on the software modernization strategy.

**Replacing**

Replacing means **switching to another package solution** that matches your business requirements. Sometimes it's more effective to switch to another system than modernize the current one.

In case of replacing, you can't reuse the existing business logic, though,—but some level of re-engineering, customization, and rewriting business logic may be involved.

Still, customizing a package can be very complex—and very risky. Just like rewriting, it should only be considered when maintaining an old legacy system becomes more difficult due to staff or hardware limitations.

**Rehosting**

Stands for **moving the system to a new physical, virtual, or cloud infrastructure** without any changes in code, features, or functions. For example, it may make sense to move an old system to a cloud or SQL-based environment.

* **Migrating to cloud** makes the system more flexible than on-site hardware and improves data security, stability, and continuous updates.
* **Moving to SQL-based x86 architecture** systems results in lower purchase costs and less strict requirements to cooling and space.

The rehosting strategy is faster than re-engineering (which may take years to finish) and comes with lower cost and fewer risks. Yet it keeps the business logic intact, which means the system works in the same way as it did before.

**Replatforming**

Developers leave the software nearly as it is, **preserving its functionality but adapting to a new platform**. The goal here isn’t to completely rebuild apps—we have the rebuild approach for that. Instead, systems are altered until they can run in the target environments like cloud-ones.

Developers take an existing system component and move it to a managed service (like a database service), making no changes to the business logic.

For example, it may imply the switch from commercial databases to horizontally scalable open-source data stores and services.

**Refactoring**

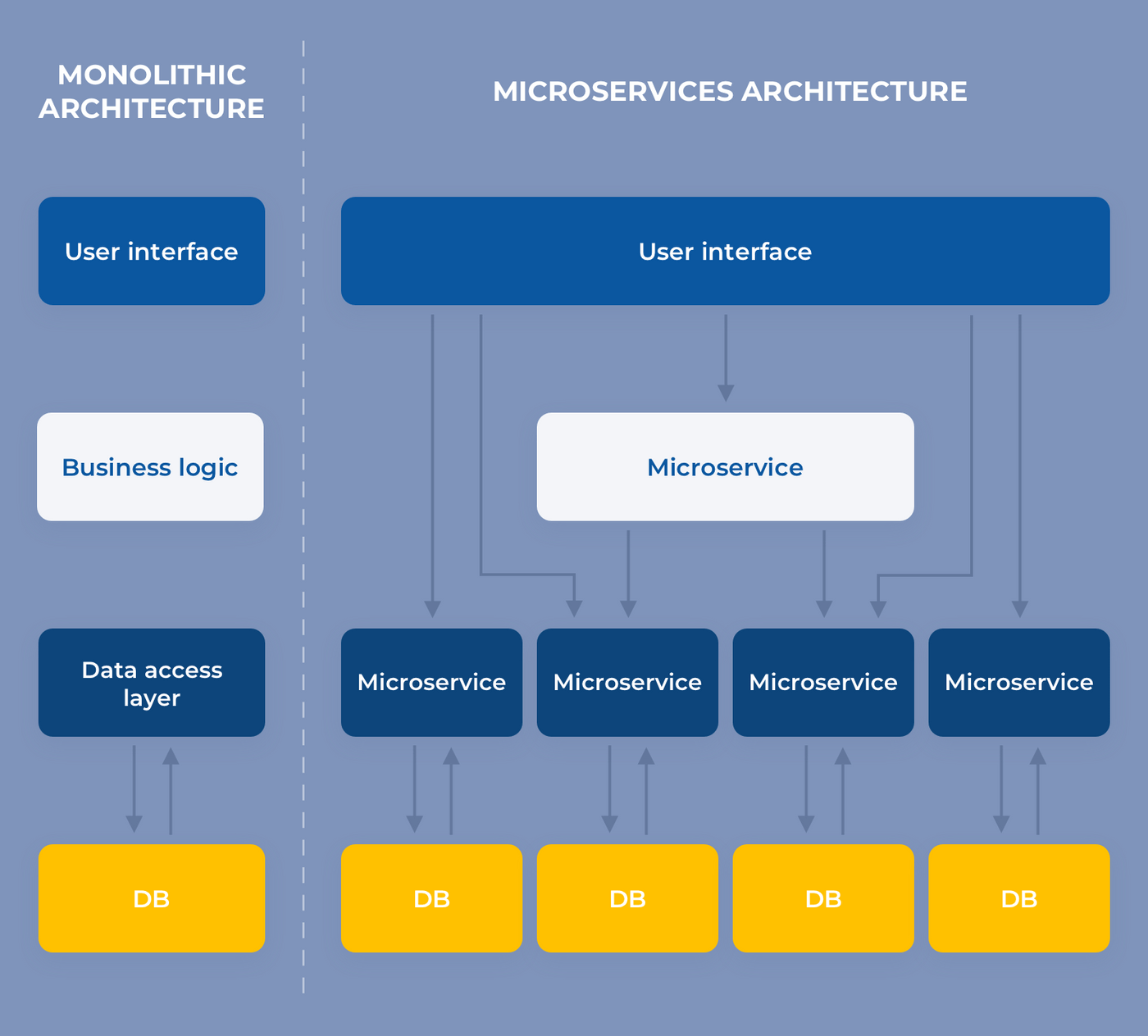
**Optimize existing code on the backend without changing front end behavior or functionality**. Refactoring is done to get rid of tech problems, improve the component’s features and structure.

By re-coding some parts of the current system, companies take full advantage of cloud-native features—and achieve max cost efficiency in the cloud.

**Re-Architecting**

Re-architecting stands for **fully changing the code of the app to move it to another architecture**. By altering the code, you can fully exploit new and better capabilities of the existing platform.

For example, you can achieve that by moving away from old monolithic architecture to modern [**microservices**](https://www.cleveroad.com/blog/benefits-of-microservices-architecture).



Legacy system modernization approaches: moving from monolithic arch to microservices

This may be a good idea, as, according to IDC IT **predictions**, **90% of all new apps** will feature microservices architectures by 2022. They claim microservices to improve the ability to design, debug, update, and leverage third-party code.

**Rewriting (Rebuilding, Redesign)**

Rewriting means **building a new system from scratch but with similar functions**.

It may be a good option when your customers want more than the current system can provide. The best justification of this system modernization strategy is when the costs of maintaining an old system have become higher than the cost of rebuilding it.

For example, your team may take a monolithic app and replicate its functionality as a cloud-native microservice running in a cloud-native environment.

While all these approaches differ, they have at least one thing in common—they require a close look at your interconnected components, systems, and applications. That's because you cannot update one component in a legacy system ignoring all the others.