WHITEPAPER

Simplify Mainframe Modernization: Navigate 8 Key Challenges with Confidence

Author: **Abhijit Vijayrao Padegaonkar**

Date: **25-June-24**

Contents

Executive summary

The IT Industry has long faced challenges in migrating outdated mainframe components to new, modern technologies. This process is often difficult, with some companies even reverting to their old systems after failed attempts.

This paper talks more about:

- The common reasons for migration failures
- Steps to take for a successful transition
- The selective use of a hybrid modernization approach

The information contained in this paper is intended for:

- Businesses planning to migrate mainframes to modern technology
- Technologists looking to modernize mainframe systems for their organizations

This will help to carve out a systematic approach towards modernization with an increased level of confidence.

Introduction to mainframe

Mainframe systems have been the backbone of computing technology for decades, evolving over time to meet growing demands. The initial mainframe system introduced in the 1940s weighed a few tons and was as big as a large room. Over time, they evolved, moving from vacuum tubes to magnetic tapes, and eventually incorporating the COBOL programming language in the late 1950s. Today, mainframes support not only COBOL but also modern languages like Java, C, C++, assembler, etc.

Thus, the mainframe super-giant has been serving business processing for more than 75 years now. Every technology has its trade-offs, and mainframes are no exception. In some cases, the drawbacks overshadow the benefits. It's time to seriously consider newer options that can meet, and often exceed, the performance of these older systems.

Challenges in the current mainframe world

Some important factors affecting the continued usage of mainframes are:

01 High operating and infrastructure cost

02 Dead, unused, or unreachable codes

03 High maintenance cost

04 Aged programs not amended for decades

05 Less user-friendly and monolithic architecture

06 Outdated programming languages and syntaxes

07 Scarcity of skilled mainframe developers and operation engineers

Brief Solution

Cloud computing, specifically Infrastructure as a Service (IaaS), has evolved significantly in recent years. This technology offers a future in computing that is fast, reliable, secure, scalable, and easy to maintain. Service providers can efficiently manage applications or data on such cloud platforms without concerns about infrastructure, security, and dependability. Many workloads are now being migrated to such cloud-native platforms, which provide highly customized setups per the requirements. So why not mainframes?

The tech industry has been working on shifting mainframe systems to the cloud. Although the cloud platform doesn't offer a complete solution for mainframes, a systematic approach can relieve much of the workload from the mainframe. As a result, the industry is making considerable efforts to migrate the mainframe workload, which indeed has become a bottleneck for business growth.
Such technology migration requires extensive planning and preparation for execution. Nevertheless, there have been quite a few cases where businesses had to revert to mainframes due to migration failures.

Common challenges in implementing the solution

Here are eight common challenges that can pose substantial risks to modernization projects. These risks should be mitigated before starting the migration process.

01 Inappropriate steps towards modernization

Velocity matters more than speed

In many cases, customers and vendors tend to hurry the migration process due to various reasons such as:

- Clients want to quickly switch to the new system to show cost savings in the financial year.
- They aim to become market leaders in the use case of mainframe modernization.
- Clients may not fully understand the complexity and repercussions of migrating mainframes to modern technology.

This often leads to inappropriate steps being taken towards mainframe modernization.

Risks

- Budget and schedule failure
- Modernizing redundant and dead code into the target system, resulting in increased complexity in futuristic maintenance
- Modernizing applications to be demised, leading to additional costs

Solution

Taking a step-by-step approach helps reduce the amount of unnecessary code that gets transferred to the new system. It also ensures that non-essential or outdated code isn't included. Follow these steps:

- Identify and assess the application to be modernized
- Remove dead code/dead components from the system/application (Retire)
- Identify functions to be retained on the mainframe (Retain)
- Identify functions to be replaced by other full-fledged products (Replace)
- Migrate remaining components into the target system (Rebuild)

02 Lack of proper assessment of the system

Introspection plays a more vital role than retrospection

A thorough assessment of the application/system to be migrated is crucial to the success of a modernization program. It gives an overall idea of the application, such as:

- Inventory details
- Call graphs
- Complexity of the application
- High-level view of the system architecture
- Database and file structures
- Interfaces and their communication paths

Risks

Without a proper assessment, you risk:

- Implementing an inappropriate solution
- Overlooking older components
- Having an incomplete view of the impact on other applications
- Budget and timeline failure

Solution

This information is enough to decide upon tentative target solutions and modernization methodology. It helps in profound planning, which reduces the risk of failure at later stages of the program. It also helps determine modernization estimation, roadmap, ROI, and similar matrices. This approach instills a high level of confidence in the customer, who would otherwise remain uncertain until the first set of modernized applications is delivered.
There are three approaches to assessing the entire application: tool-based, questionnaire-based, and hybrid. A long-term vision for the application's status is crucial in decision-making. For instance, if an application is expected to be retired before reaching its ROI or breakeven point, migration may not be a viable business option. Such vulnerabilities can be effectively identified during the assessment phase.

03 Lack of subject matter experts (SMEs)

There is no need to reinvent the wheel

Many technical and functional SMEs are already tied up with ongoing projects, making them unavailable or partially available for the modernization project. Going ahead with a modernization project without a dedicated subject matter expert (SME) with comprehensive system knowledge is a classic recipe for disaster.

Risks

- Directionless project execution in silos
- Potential gaps in documentation, development and quality assurance (QA)
- Identification of issues at a very later stage of the project

Solution

SME knowledge is required at every stage of the modernization program, whether it's manual rewriting or automated code conversion. Their experience working on the existing system helps in replicating the functionality on the target system. The SMEs' functional and technical knowledge of the system helps design and plan better. They can also help validate the outcome at every stage of modernization. Therefore, anything that is missed can be corrected before it greatly impacts the overall modernization program. Hence, a dedicated technical and functional SME is a must in all modernization programs.

04 Lack of documentation

The faintest ink is sharper than the sharpest memory

As software and business solutions have evolved over the decades, the processes surrounding them have, too. However, mainframe systems are older, and comprehensive documentation is often lacking. This has long been a challenge in mainframe development programs, as many systems do not have up-to-date functional specifications and units, nor do they have integrated test cases.

Risks

- A fragmented and incomplete knowledge base
- Increased risk for future maintenance projects

Solution

As part of the modernization process, creating functional documentation and unit test cases/results during the reverse engineering phase is crucial. This helps test the application in the target system and creates the base for integration testing. Functional documentation is created as part of the reverse engineering mentioned above. Test cases can be created by testing the modules in the development region. SMEs should validate all the documentation created during this phase.

05 Lack of awareness of the current architecture

Self-realization is immense power and energy

Mainframe systems are large and often contain decades of technical debt due to their complexity and numerous interfaces. Knowing how these interfaces integrate with the core mainframe system is paramount for smooth migration. The more interfaces involved, the more complex the migration becomes.

Risks

This complexity can affect the sequence and planning of application modernization.

Solution

Knowing the current architecture inside out plays a pivotal role in migration decision-making, such as:

- Application migration priority and sequencing
- Pre- and post-migration interface connectivity
- Communicating with and from applications retained on the mainframe
- Service-level agreements (SLAs) to feed and get feed from the interfaces
- Impact of mainframe infrastructure cost on applications retained on the mainframe
- Overall timeline to migrate to the new system

06 Insufficient time for reverse engineering

Take time to sharpen the axe

The mainframe may have very old components that have not been updated for decades. In some cases, only the load module is available, and the source code is not. As COBOL, a procedure-oriented language, is the primary language for mainframe development, mapping it with current object-oriented, open-source technology is a huge challenge. Moreover, for any other migration, there are at least a few commonalities between the source and the target, making it easier to migrate. However, in the case of mainframes, the source and target are at the extreme ends of the technology. Thus, there is very little scope for any error in reverse engineering.

Risks

- Generating incorrect documentation
- A cascading effect of incorrect documentation throughout the project

Solution

Mainframe migration is different from any other migration. Hence, the thumb rule used to estimate other migration projects cannot be used for reverse engineering estimates. Eventually, reverse engineering must be estimated properly by referring to the parameters such as:

- Types of components
- Complexity
- Lines of code
- Number of interfaces
- Criticality of the application to be migrated

Taking a certain percentage of the forward engineering estimates for assessing reverse engineering is not a good idea. Guesstimating based solely on the target system is not feasible; the legacy code is the only complete and functional source that defines the overall system. The target system must be derived from this code, so reverse engineering requires additional time and focus. However, applying agile methodologies to reverse engineering is often debated. Agile works well in a known system, but this is rarely true in mainframe modernization projects. Therefore, delivering story points based on functionality, lines of code (LOC), complexity, or the number of programs is often imprecise.

Accurate estimates can only be made after a thorough system assessment, as described in point 2 above. At best, estimates can be refined if technical and functional SMEs are available and information-rich documentation is accessible.

There are multiple approaches in which reverse engineering can be planned.

01

Batch flow-dependent:
Reverse engineering is carried out according to job flow. Thus, the jobs, programs, transactions, and other technical components encountered are reverse engineered as a single entity. This helps enhance functional knowledge as progress is made into the next level of dependencies. However, common functions may have to be visited repeatedly while reverse engineering different batch jobs.

02

Transaction-driven:
Considering the transaction flow, all technical components entering the flow are reverse engineered together as a single entity. This is a data-centric approach, and understanding the data flow improves with progress.

03

Functionality-based:
Programs and other components pertaining to one function are clubbed together for reverse engineering. One consolidated document is prepared for the same, which can be treated as a requirement document for ongoing projects.

A suitable approach must be brainstormed after the assessment is done. It depends upon the architecture of the system.

Typically, a technical developer can perform reverse engineering as per the following matrix.

Simple COBOL program	600 Lines per day
Medium Complex COBOL program	500 lines per day
Complex COBOL program	350 lines per day

Note: Complexity is a parameter that changes from case to case

07 Lack of parallel testing

Well-tested products always catch customers' eyes

Testing plays an important role, especially when the entire system is being migrated. For any regular development project, testing is only done for the developed and impacted code. But that is not sufficient for such a huge transition project. Mainframe applications are integrated with multiple upstream and downstream applications. Hence, end-to-end testing must be done, including all interfaces.

Risks

- Tendency of functionality failure
- Identification of bugs in a higher environment

Solution

Parallel testing must be planned to ensure the correctness of the testing, i.e., testing on the mainframe and the target system. The results of such an intermediate testing phase give confidence to all technical and business stakeholders. Any defects identified during this phase must be fixed per the incident life cycle. It's recommended that parallel testing covers at least three month-end processing cycles.

08 Monitored usage of automatic code conversion

A balanced use of accelerators and brakes makes it a pleasant drive

Clients always want to migrate quickly, and often have a very ambitious timeline. Thus, tools that convert COBOL code into the target language are preferred to meet the timeline.

Risks

If these tools malfunction, they can cause major setbacks.

Solution

It’s important to monitor the use of automatic code converters. Consider the following:

- The tool processing cost is based on the number of lines in the code. Hence, only codes filtered through the process mentioned above should be fed to the tool for conversion.
- The programs fed to the tool must be simple. The tool’s code conversion efficiency is 60%, so analyzing the tool output for simple programs becomes easy.
- Technical and functional expertise is required to convert complex codes, preferably manual rewrites.
- The tool performs a technical conversion of the program, resulting in a like-for-like transformation. The output offers no additional benefits over the target system.
- Functional SMEs’ availability to validate the tool’s outcome is key to the success of code conversion.

For complex codes, a hybrid model that combines manual rewriting with automatic conversion for simpler programs is often the best approach.

Conclusion

Mainframe systems are inherently complex and modernizing them adds an extra layer of challenge. Upgrading outdated technology takes time and requires careful planning. Delaying modernization only makes the process harder. Therefore, starting early is key to a successful transition. While the challenges mentioned above are significant, they can be addressed with thoughtful and well-structured actions. The following steps can help ensure a smoother modernization process:

Retire:

Review the system to identify old, unused components and remove them. This immediately reduces storage needs and simplifies the migration.

Retain:

Identify critical components that need to remain on the mainframe, particularly those handling core processing. Remove these from further analysis, focusing only on their data connections. These components can undergo in-place modernization.

Reuse:

For parts of the system that can be reused, migrate them to the cloud. Cloud service emulators can support this transition.

Replace:

Where possible, replace older functions with more advanced products that fit current business needs. Establish connections between interfaces and set parameters for compatibility.

Rebuild:

After reducing the system's complexity through the steps above, the remaining components can be migrated to the new system, either through manual rewrites or automated code conversion.

A successful modernization requires the expertise of subject matter experts (SMEs) who understand the system's entire functionality. Their insights are essential in determining which components fit into each of the above categories.

A thorough system assessment will reveal the technical specifics, such as component complexity, dead or unreachable code, and program and data flow diagrams. Discussions with SMEs and analysis of the outcome will also help generate the technical architecture diagram.

Effective use of available documentation and functional discussions and sessions with SMEs will help generate functional flows, data flows, and overall business knowledge.

After reducing the inventory, manual rewrite or auto code conversion tools can generate the code in the target system. With reduced inventory, the pros and cons of both methodologies are greatly diluted.

Parallel testing of the entire application for at least three months ensures the correctness of the result. It adds overhead to the budget but is negligible compared to the advantage. Allow three months to execute month-end cycles to fix any outstanding issues.

Another advantage of taking such small, healthy steps is that the business can reap the benefits after completing each step. This is called "functional modernization," as "functionality" is at the epicenter of all decision-making. Various types of mainframe modernization that can be considered are:

Technological modernization:

This includes upgrading components like CICS, modernizing batch processes, or rehosting all applications outside the mainframe using an emulator

Storage modernization:

Moving data storage to the cloud can significantly reduce costs

In-place modernization:

Applications can be modernized while still on the mainframe by upgrading technologies like compilers or enhancing system compatibility and performance

APIfication:

Making mainframe programs accessible through APIs can extend their usefulness and increase flexibility

By following these steps, mainframe modernization can be a well-planned, gradual process that leads to long-term success.

LTIMindtree