Mphasis
The Next Applied

Modernization

POV by Suresh Nair, Vice President & Principal Consultant

Contents

1.
Introduction

‘We have an existing application, with logic we like, but on a platform we don’t like. Can you get that logic onto a ‘better’ platform? We refer the activities that attempt to do this (i.e., moving program logic from one technology to another) as modernization.

Statistics show that modernization projects have a very poor success record. The migrated applications often end up being ‘white elephants’, very expensive to maintain and providing little business benefit over the applications they replace. As a result, they are often perceived as wasteful IT overhead.

This document talks about the main reasons these projects fail, and what we can do to ensure they succeed and benefit both business and IT.

2.
Using an analogy to reframe the problem

Imagine a line drawing of a person’s face that has been worked on for 20 years. Over the years, we have had the time to correct all the errors, so the drawing is amazing likeness. With time, the person has grown older, but the drawing has been updated to add age lines and wrinkles to keep the picture ‘current’.

Now, another artist has been asked to recreate the same drawing, but in chalk instead of pencil. The artist wanted to look at the subject, but he was asked to copy from the old drawing that was created in 20 years.

A copy of a drawing is a copy of a drawing; it is not a portrait of the subject. The technique for creating a chalk drawing works very differently from the technique for creating a pencil drawing. While we did get rid of many of the mistakes, there are a few error lines still in there, as well as smudge marks from the corrections.

If we look at the artist at work, we can see that their technique is different. However, he has been asked to throw away his years of experience, and instead mimic the pencil style using the chalk.

At this point, any artist worth their salt would throw their chalk and walk away. However, if the starving artist does continue gamely to the end, we can pretty much predict that the resulting work will (a) not be a good likeness of the subject (b) be a horrible chalk drawing and (c) be impossible to keep 'current' like we did for the old pencil drawing.

This is exactly what happens in most modernization projects. The developers are not allowed to talk to the business to understand the context or usage; they are instead asked to migrate logic line by line. The migrated applications are not suited for the target architecture and are very hard to maintain.

The mistakes are the same in both cases -

1. The style or patterns do not fit the new medium: While the logic works, it is not optimum for the target architecture. Well-designed COBOL and Java applications have completely different approaches to how data is modeled and managed, which means the way the application uses the data is also designed differently. In COBOL, shared data structures are described in copy books, which are used through a data flow. In Java, we break the data flow up into modules by 'encapsulating' data into objects. These are diametrically opposing methods for modeling and sharing data. We can mimic COBOL copy books on Java and migrate the COBOL logic to use this, but the resulting application would be considered a very poor Java application, which is not intuitive for a Java resource to maintain.
2. We are copying the past mistakes as well as the things that worked well. It is really difficult to migrate and unwind past mistakes at the same time. This means that there is no significant improvement immediately after a legacy migration. It takes a couple more iterations before the overall system improves.
3. In the example, chalk and pencil artists use very different techniques. In the same way, the process of capturing requirements and building technology solutions against those requirements is very different for different technology platforms. The migrated application is not ideally suited for the build style of the target technology, resulting in a build process that is not efficient. Due to this, the productivity drops post migration.

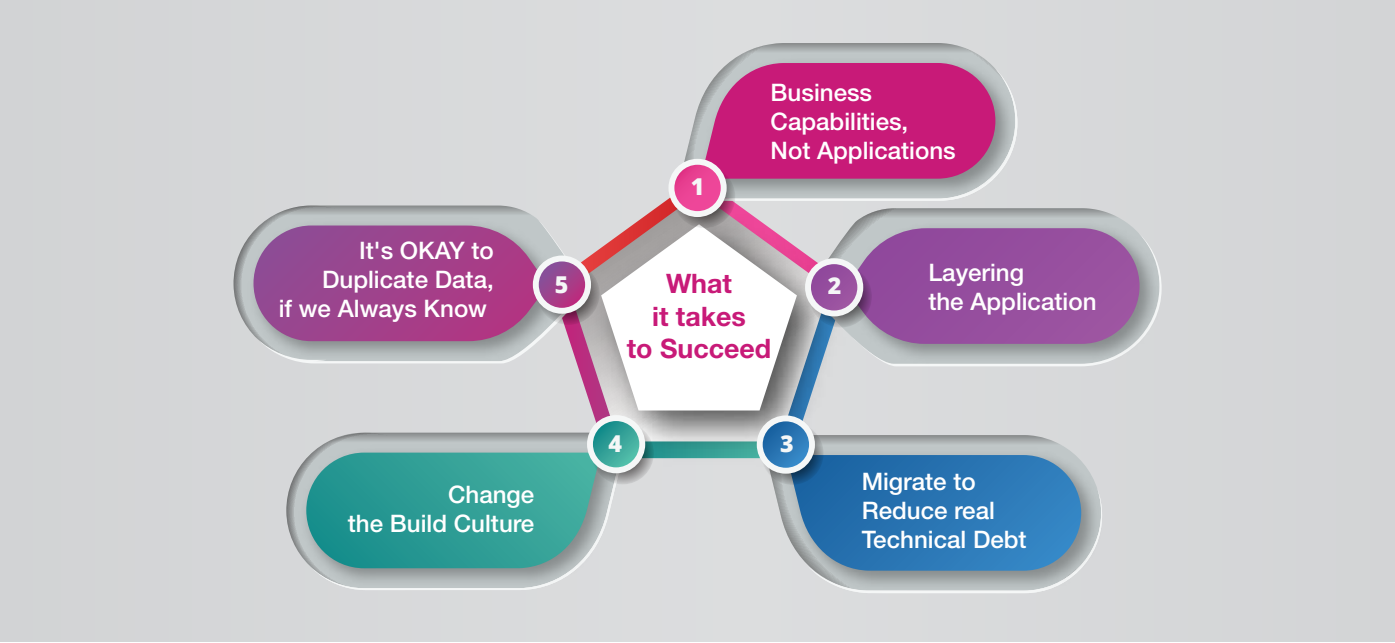
In the same analogy, there is a way to get a great chalk drawing. An experienced artist will ignore the directions given to him to mimic the line drawing. Instead, he will step back and use the pencil drawing to build a mental image of the subject, filling in the gaps from experience. He will

create the image from scratch following their mental model, rather than the actual lines in the pencil drawing. Now, with a quick glance at the subject, they can quickly pick up what they might have missed and fill in the details.

This is very close to what we do to ensure that modernization projects succeed.

3.
What it takes to succeed

The approach to ensuring modernization projects succeed is built on a few key changes in thinking:



Business Capabilities, Not Applications

Traditionally, we have focused on applications, migrating single or groups of applications at a time. These applications provide a set of ‘business’ capabilities, which are used to record and automate different aspects of the business. Usually, a single business process involves making business capabilities from different applications work together.

We have found that focusing on business capabilities instead of applications helps us handle the migration better. This is the technology equivalent of the chalk artist creating a picture of the subject in their mind's eye. The visible business capabilities in the applications show us how the technology is being used today. By focusing on these instead of the individual lines of code, we have a much better understanding of what business needs. We can now figure out how to deliver that capability in the new technology.

If we break these capabilities down the right way, we can often find off-the-shelf or existing logic that can be reused or repurposed with much less effort than a full migration. This ‘right way’ usually means:

a) Breaking down applications to organizational boundaries
b) Encapsulating functionality that has the same ‘rate of change’

Let us take a practical example to understand this better.

Imagine that we have a small company with a few departments. Departments can order things they need through a procurement process. Different teams take care of the actual procurement based on what is being ordered. All procurement activities involve finance. We need to ensure that the departments are operating within their budgets, and that we pay the vendors on time.

If we think of buying chairs for the office vs requesting compute capacity on the cloud, it is going to complicate the procurement request and fulfillment process. The office supplies department and the IT infrastructure departments represent two logical organizational boundaries. We therefore need to treat these as two different business capabilities. The financial reporting for both of these actually is almost identical. Both procurement processes could invoke a single business capability to make the general ledger entries against the procurement.

Layering the Application

When splitting out business capabilities, we often think of these as 'vertical' functional slices; however, we also need to deconstruct the ecosystem 'horizontally' into layers. Our core systems of the past often combine systems of records, process control, rules and user interfaces into single systems. We recommend layering these into:

1. Systems of record: System of record layer is responsible for storing the current 'state' of critical resources which is important for running the business. Think of where we would go to answer these questions: Who are your employees? How many goods do we have in stock? For a bank, how much do we owe the customers who have deposits with us? How many loans do we have?

2. Systems of interaction: System of interaction provides the capabilities that users (or other systems) see; it is used to initiate changes and track ongoing processes. Users could capture details of something that happened (e.g. customer moved to a new house), trigger a process (e.g. customer deposits a check to their account) or perform a manual task (e.g. authorize a non-standard transaction).

3. Systems of intelligence: System of intelligence has the logic to interpret the events from the system of interaction and manage the processing of these events through to completion, updating the systems of record as we reach milestones in the process.

By enforcing this 'horizontal' separation into layers, we significantly simplify the migration of any individual layer. Past experience has taught us that many of the 'mistakes' in past systems are often resolved by this separation of layers. It formalizes the interactions between systems in the same way that is intuitive to business and system designers.

Migrate to Reduce Real Technical Debt

Problems with existing systems are often expressed as 'technical debt'. Wikipedia defines technical debt as 'the implied cost of additional rework caused by choosing a simple solution now instead of using a better approach that would take longer'. For instance, in the past, we may have added functionality into a system where it did not fit because of the availability of resources or political realities within the organization. However, we are biased in our interpretation of technical debt. Very often, technologies that are not part of the current crop of buzz words are automatically classified as representing technical debt. This is not necessarily true.

Let's take an example of application hosted on a mainframe that is past end of life. If we keep supporting the mainframe because it is cheaper than the alternative of moving off it, we are accumulating technical debt.

However, if something is on a 'current' mainframe, well supported by a reputable vendor, do we need to replace it?

Not really; it is not incurring technical debt. If the logic is written in a language where it becomes hard to find resources, for instance assembler on mainframe, then that part of the logic could be considered as accruing technical debt.

At a certain point in future, making changes to the logic is going to be an expense. So, while assembler on a 'modern' mainframe may not represent technical debt today, it will eventually become a significant cause of technical debt as the current development team ages out, and it becomes harder to find resources to replace them.

In table 1, we provide samples of technical debt causes, and how we remedy them.

Symptom	Fix
Simple, limited changes to process flows cannot be released independently. Instead, they must always be coupled with infrequent, large code releases.	Pull process logic out of core systems into a separate layer.
Infrastructure on which an application is running (hardware, operating system, container) is nearing, at or past end of support from the vendor.	Migrate the logic to platform with a longer lifespan. Ensure that the organization internalizes the accelerated obsolescence cycle for modern technologies.
The 'shape' of an application spans multiple departments, requiring consensus for any changes. Especially a problem when the co-owners have disproportionate control on the roadmap.	Move towards a microservices architecture, with care being taken to break at organizational boundaries. Separate out foundational services from business logic.
Faults in non-functional areas: E.g. a shared computing resource (memory, CPU, hard disk) maxes out occasionally, causing system outages.	Break down processes to pull out tasks that do not need to be on the execution path. Use modern stream technology to have these happen close to real time; but on separate hardware.

Table 1: Some examples of technical debt

Change the Build Culture

Once the business capabilities are migrated to the new platform, a team would be required to support the newer version. There is a 'who' and a 'how' question that needs to be answered here. Let us start with 'who' first.

To build great technology solutions to difficult business problems, teams need at least two skill sets: (1) an experience with the business and (2) experience with the technology. If we bring a new team, we lose (a), if we use the existing team, we lose (b).

The second is 'how'. Better alignment between business and IT is achieved through agile. Faster turnaround of individual IT projects is achieved today through the use of 'DevOps'. Increased stability and overall team productivity are achieved through test-driven design, and the team should follow the below practices.

What we have seen is that introducing these on the legacy platforms prior to the start of the transformation project helps with both 'who' and 'how' parts of the transformation. Successful transformation is dependent on some restructuring of the legacy platform. By introducing DevOps, agile and test-driven design in the legacy platform, the existing team improves upon their skills and achieves a productivity lift. They get acquainted with the transformation and see it as an opportunity rather than a threat. This goes a long way towards making modernization projects succeed.

It’s OKAY to Duplicate Data, if We Always Know We are Doing it

One of the major causes of delays in any kind of technology transformation is the refactoring of data structures. It was a tenant of good design for nearly 30 years that data should not be duplicated across systems. This means that all the teams that want access to a piece of data have a say in how it is structured. Building consensus on these data changes kill the momentum of most large projects.

With the increasing popularity of microservices, there is a push to think beyond the traditional model. Microservices need to be independent, which means we cannot have data dependencies across different functional units. Duplication of data is considered acceptable in this world, provided we understand how and why data is duplicated.

The most popular analogy that can be used here is to think of the business components as pipes, and databases only as reservoirs. We can ‘move’ data from one reservoir to another, and this is how things work. E.g. ‘customers who want to open accounts’ is one reservoir, while ‘customers with open accounts’ is another. The business components involved in account opening – identity checks, credit checks, relationship and account type selection – are ‘pipes’ that come together to move the case from one reservoir to the other.

Prior to a legacy migration, it is very beneficial to go through the effort of deconstructing the application into microservices, with data sharing issues resolved. We convert shared data into handoffs of data between systems. These APIs provide clarity on how data ‘owned’ by one system is consumed by others. Within each of these microservices, we can now evolve data structures independently, using the APIs to isolate other systems from internal changes.

4.
The 4Rs for success in modernization

Putting all of these changes together, we have a repeatable model for modernization.

As we discussed at the outset, modernization is one activity within a larger program where we have decided to move existing logic from one technology to another. Typically, IT and business should work together to create an inventory of the applications across the enterprise and identify actions to be taken for each one. A common classification they arrive at, would be something like this:

Modernization addresses the changes in the ‘renew’ and ‘replace’ buckets.

For each of these, we recognize that there are several things going on at once:

1. We are addressing the needs of the business, which are changing. Rather than just focusing on migrating existing logic, we need to look at the desired end state.
2. Deconstruction of the existing application into layers and microservices prior to extensive re-platforming of logic
3. Put in place the processes and skills to sustain the new platform. This requires retraining of existing resources on the new technologies and training new teams on the business.
4. We need to create a DevOps ecosystem to automate development tasks as much as possible and support technology

   When we look at any application we are transforming, we find that all these activities are happening in parallel.

5.
Planning the migration

Putting all of these changes together, we have a repeatable model for modernization.

As we discussed at the outset, modernization is one activity within a larger program where we have decided to move existing logic from one technology to another. Typically, IT and business should work together to create an inventory of the applications across the enterprise and identify actions to be taken for each one. A common classification they arrive at, would be something like this:

While the exact steps and sequence is very organization and context specific, we see the following tasks in successful migration plans:

1. We take the inventory of applications to be migrated, and break this further down into microservices
2. We further split the application into layers (system of engagement / system of record / system of intelligence) and place the microservices into one of the layers. If required, split up the microservices further.

3. Finally, look at any shared data structures across microservices, and break the links till only one microservice owns the data. Create data flows to show the flow from service to service for the business capabilities.

4. We gap this list of microservices against the inventory outside of the migrated application, flagging for eliminating all the microservices that are (or will be) available elsewhere

5. We now have an inventory of the microservices that will exist in the end state; and a 'from' and 'to' map for each of the business capabilities, showing how it will move from the legacy platform to the end state microservice

Initially, even before we start on the migration, some of these microservices can be created 'in situ', by wrapping the legacy code. Modern tools exist that make this much easier. We can now do a technical debt assessment for each of the business capabilities and decide which ones can be left on the technology they are on, and which ones really need to move.

What we have seen arising from this exercise is:

1. A large percentage of the legacy application will not need to be migrated, as capabilities are either dropped or moved to existing capabilities in other modules

2. 'Dead' code is identified and eliminated from the scope. Since we are focusing on business capabilities, it becomes much easier to see functionality that is not relevant for the current business operations.

3. We can see a way to delivering business benefits almost immediately after starting the program. The breakup into business capabilities and layers frees up the logjam between functionalities and allows the processes and business capabilities to evolve at different speeds. This leads to increased business buy into the program.

4. We are taking most of the existing IT community along with the change, upgrading their skills and sustaining their importance to the business. This helps withstand the run of the overall program.

5. We remove many of the root causes of slowdowns in migration projects – arguments about data and process flows – avoiding overruns on the overall project

In parallel, the following would be happening to the development teams:

1. If not already practiced, introduce Agile. The changes required to package existing capabilities into microservices would be converted into epics and stories, which can be picked up in scrum meetings along with BAU work.
2. If not already practiced, introduce DevOps. Automate the environment setup and build, increase the use of automated continuous integration and certification, and mandate the use of unit tests as gatekeepers to ensure developer accountability. Synchronize legacy releases with modern platform releases, so teams start adjusting to the new turnaround times for changes.
3. If not already practiced, move towards test-driven design, which encourages the creation of well-bounded capabilities, without data dependencies. There should only be dependencies between business microservices and infrastructure microservices.

6. Mphasis tools and accelerators to help in modernization

at Mphasis, we have a complete suite of accelerators, pre-integrated third-party products and best practices to implement all of the steps outlined in this document. Some of the tools are:

Front2Back™

Provides a set of tools to accelerate the 'vertical' separation of applications into 'system of engagement', 'system of record' and 'system of intelligence'. Implements out of the box approach for most of the tools (machine learning, dynamic schema mapping, conversational context tracking) that make modern applications much better than legacy applications. It limits what is needed from legacy microservices and provides a platform where logic that is being pulled out from legacy platforms can be hosted for the future.

XRAE

Provides a comprehensive suite of pre-integrated tools to increase the productivity of developer communities.
It combines the continuous integration ecosystems with technical debt measures, so we can continuously improve the productivity of the team while iteratively reducing technical debt within the application.

Cloud Migration Frameworks

Where the target for a migrated platform is a cloud environment, we leverage a variety of cloud migration frameworks for platforms like PCF, Azure, Google, Amazon, Force.com, etc.

Automated Code Migration Services

We partner with a number of automated code migration tools and service providers for the automated conversion of code between platforms. These are aligned to the approach outlined here, in that we deliver cloud-ready microservices from legacy code.

Next Gen Transaction Processing

Modern high-performance transaction processing ecosystems use ‘immutable data’ and ‘functional programming’ instead of the more traditional ‘distributed transaction management’ frameworks like CICS or TIBCO, and record level locking.

7.
In conclusion

Modernization projects have a reputation for being hard, prone to overruns and having a low probability of success. By understanding what it takes for the end state to be successful, we can achieve consistent success. Using the tools mentioned in this document, we can significantly reduce the time required to achieve the desired business end state, eliminate the common causes of project delays and increase the satisfaction of all stakeholders in the process.

About Mphasis

Mphasis (BSE: 526299; NSE: MPHASIS) applies next-generation technology to help enterprises transform businesses globally. Customer centricity is foundational to Mphasis and is reflected in the Mphasis' Front2Back™ Transformation approach. Front2Back™ uses the exponential power of cloud and cognitive to provide hyper-personalized ($C = X2C^{2}_{TM} = 1$) digital experience to clients and their end customers. Mphasis' Service Transformation approach helps 'shrink the core' through the application of digital technologies across legacy environments within an enterprise, enabling businesses to stay ahead in a changing world. Mphasis' core reference architectures and tools, speed and innovation with domain expertise and specialization are key to building strong relationships with marquee clients. To know more, please visit www.mphasis.com

For more information, contact: marketinginfo@mphasis.com

USA
460 Park Avenue South
Suite #1101
New York, NY 10016, USA
Tel.: +1 212 686 6655

UK
88 Wood Street London
EC2V 7RS, UK
Tel.: +44 20 8528 1000

INDIA
Bagmane World Technology Center
Marathahalli Ring Road
Doddanakundhi Village
Mahadevapura
Bangalore 560 048, India
Tel.: +91 80 3352 5000

www.mphasis.com

VAS 17/05/19 US LETTER SIZE BASIL 5280